**Docker Developer Project Assignment for CTF Challenge Creations Report**

**Creator: Karthikaa R**

**Overview**

This report outlines the creation and implementation of four CTF challenges. Each challenge demonstrates a common web vulnerability: SQL Injection, Cross-Site Scripting (XSS), Command Injection, and Insecure File Upload. The challenges are hosted in Docker containers, ensuring a consistent and isolated environment for participants.

**Docker Setup**

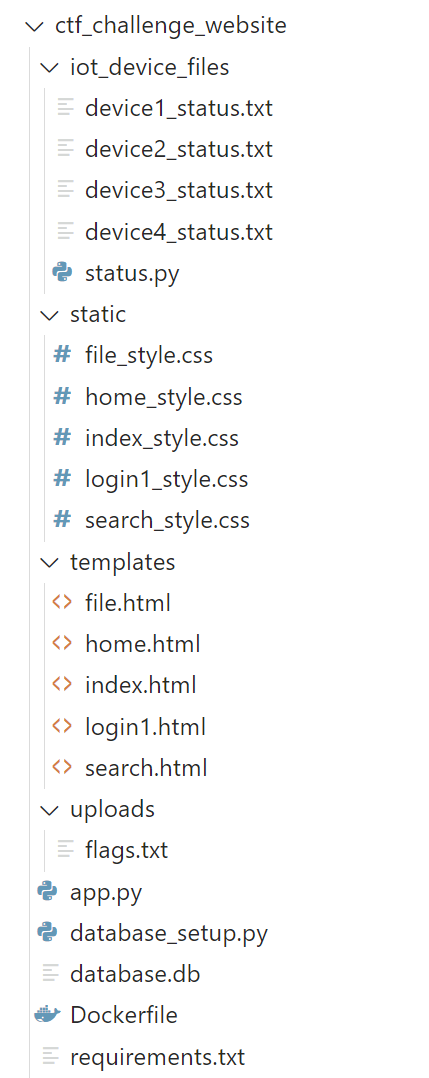
**Environment**

I am using Selfmade Ninja Academy (SNA) lab's Docker environment to build a straightforward CTF challenge.



**Folder Structure**

The folder “ctf\_challenge\_website” contains all the necessary files for the CTF challenges. Below is the structure of the folder:



**Docker Commands**

To build and run the Docker container on port 2222, use the following commands:

**Bash**

# Navigate to the directory containing the Dockerfile

cd ctf\_challenge\_website

# Build the Docker image

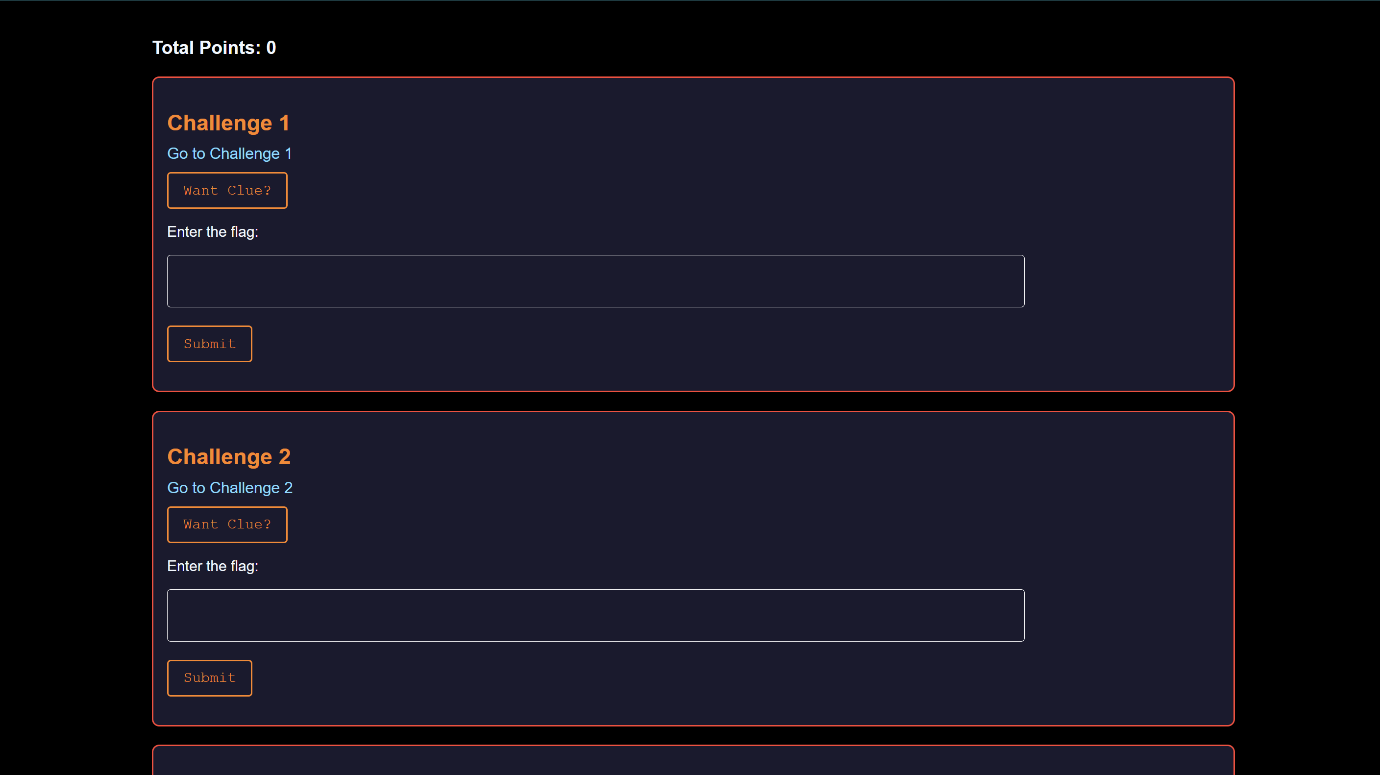
docker build -t ctf\_challenge .

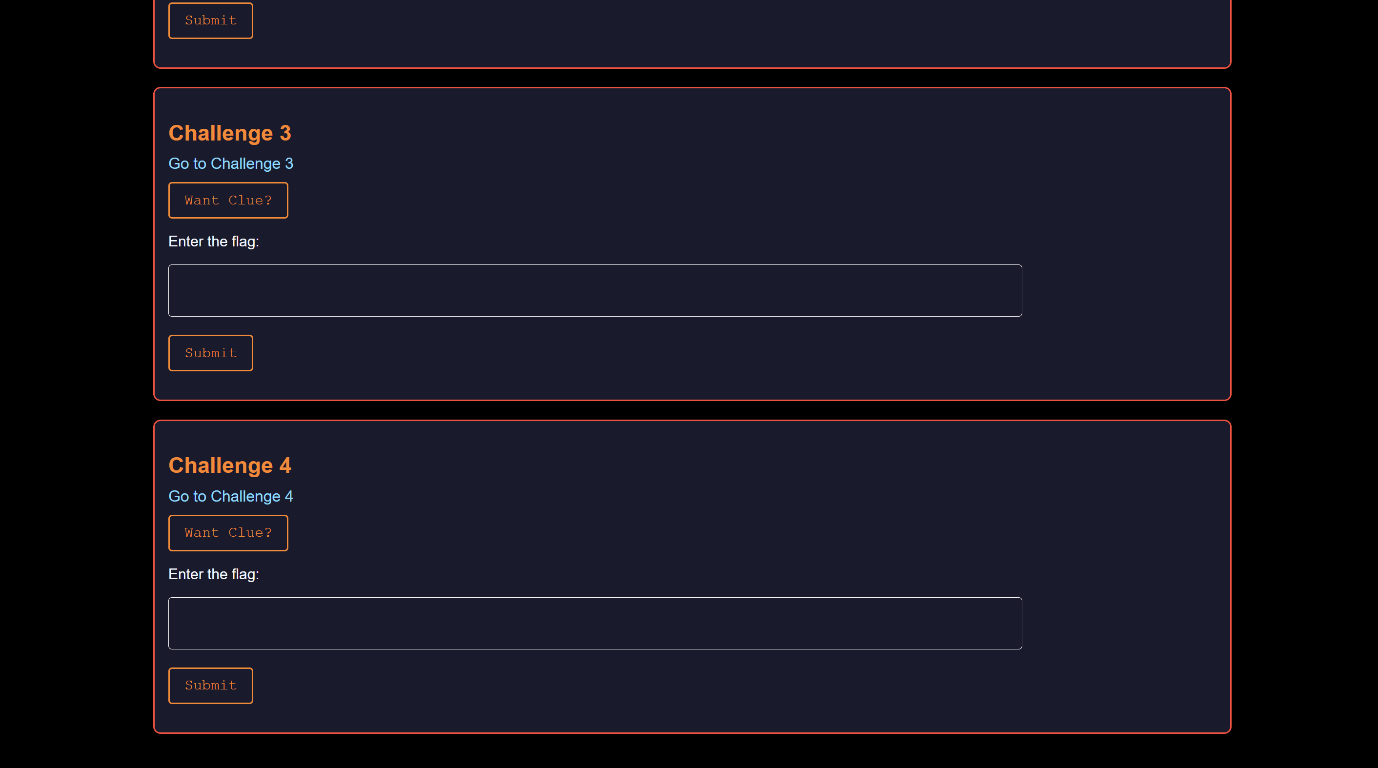
# Run the Docker container on port 2222

docker run -d -p 2222:2222 –name ctf\_challenge image

**Challenges**

**HOME PAGE**





**Challenge 1: SQL Injection**

**Description**

This challenge involves bypassing a login form using SQL Injection.

**Clue**

"Why not cheat the database? Think creatively."

**Code and Vulnerability**

The vulnerability lies in the way the SQL query is constructed in app.py:

**python**

# Vulnerable code in app.py

username = request.form['username']

password = request.form['password']

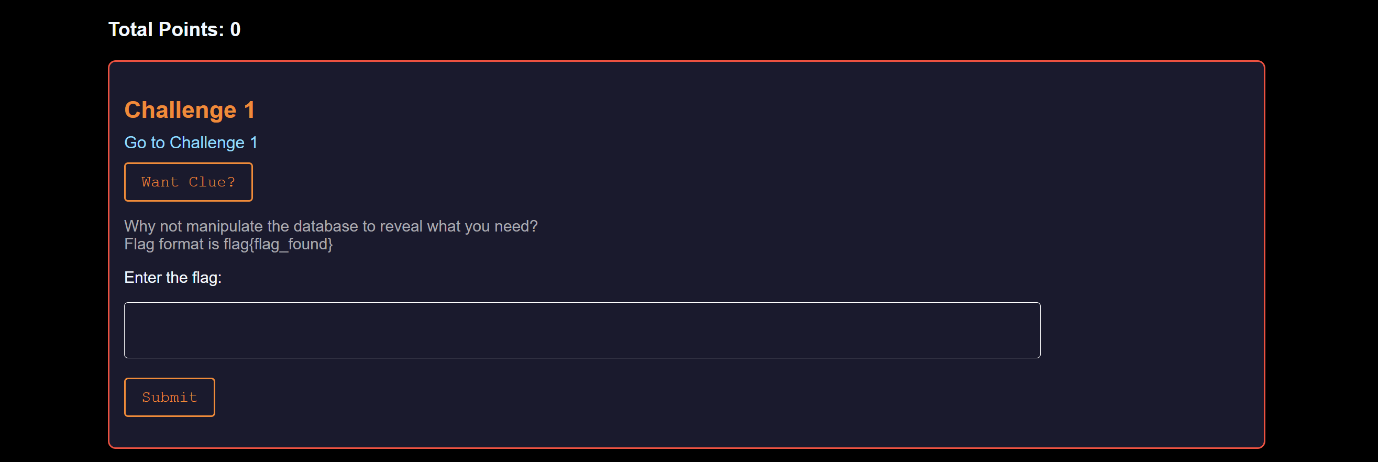
query = f"SELECT \* FROM users WHERE username='{username}' AND password='{password}'"

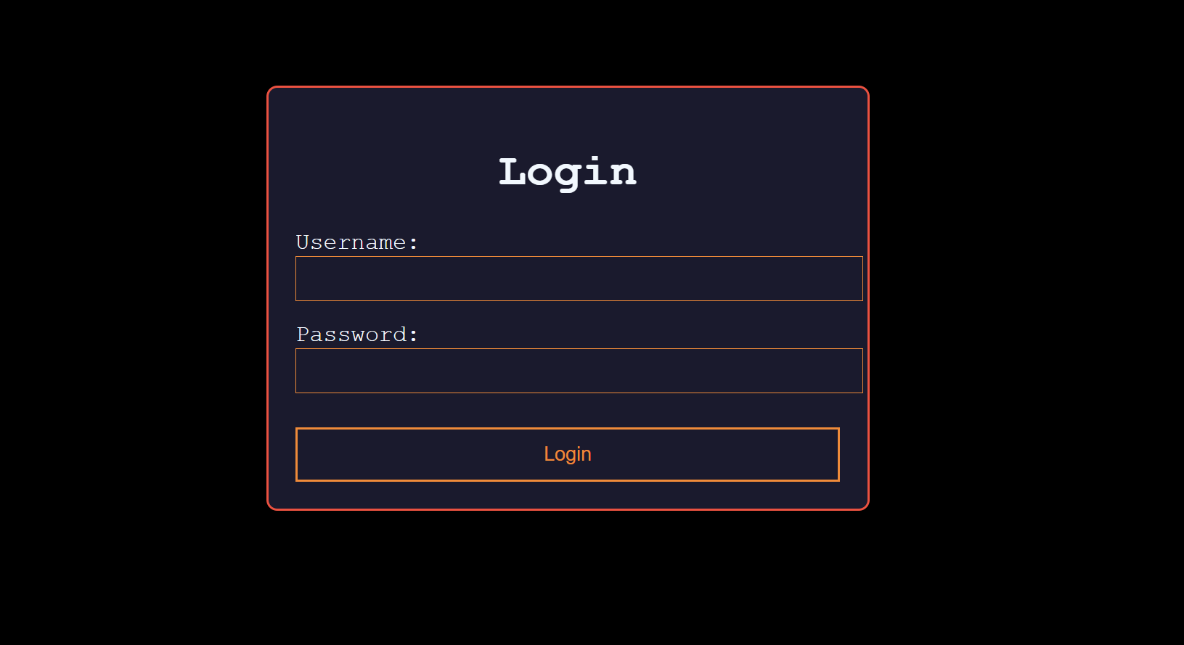
result = db.execute(query)

The query directly includes user input without proper sanitization.

**Steps to Solve**

1. **Navigate to the login page:**

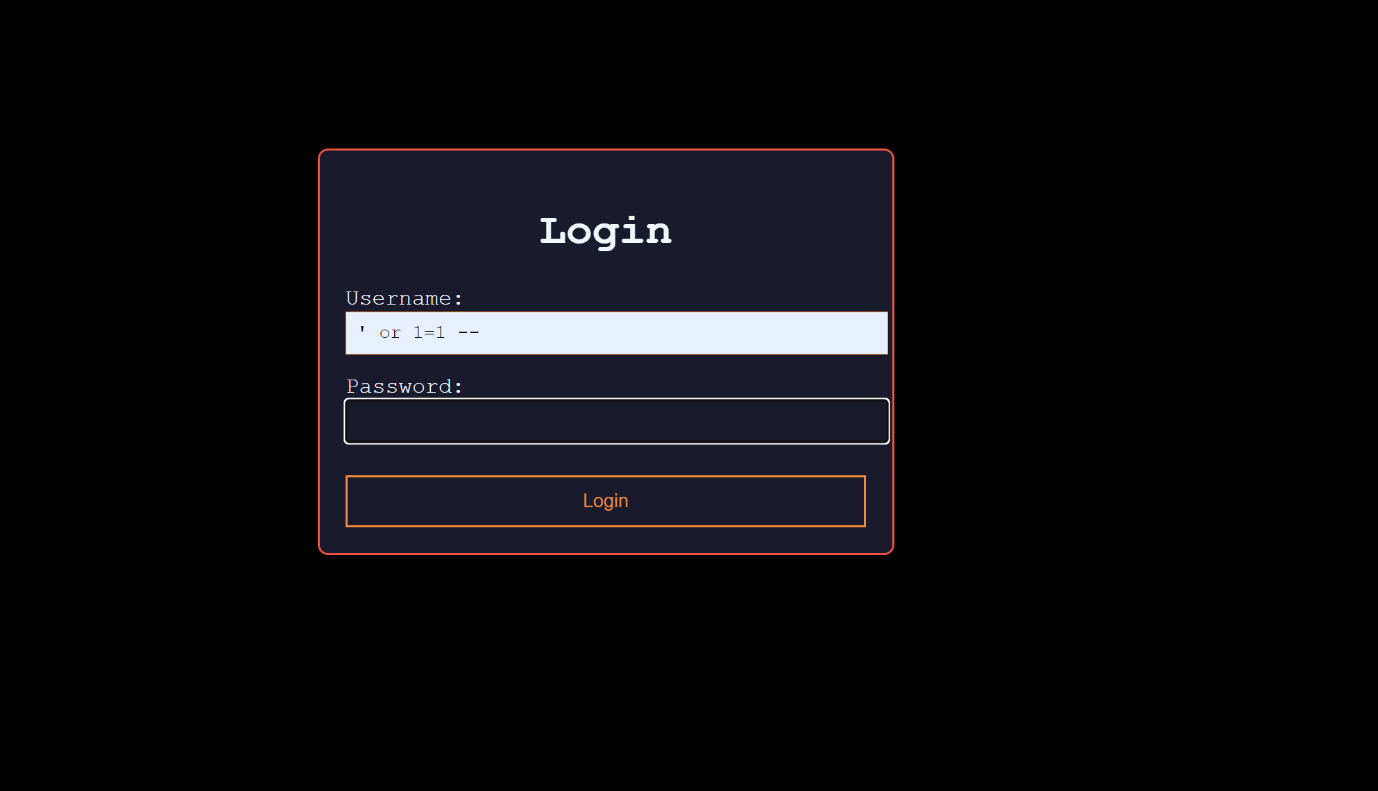




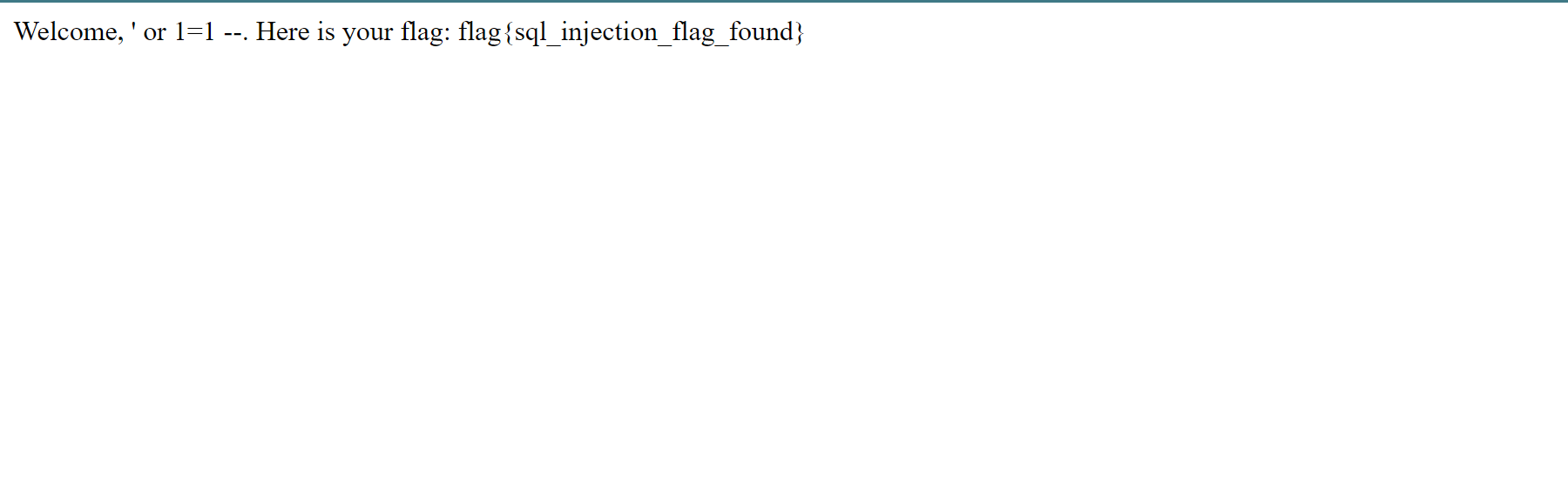
2. **Enter SQL Injection Payload:**

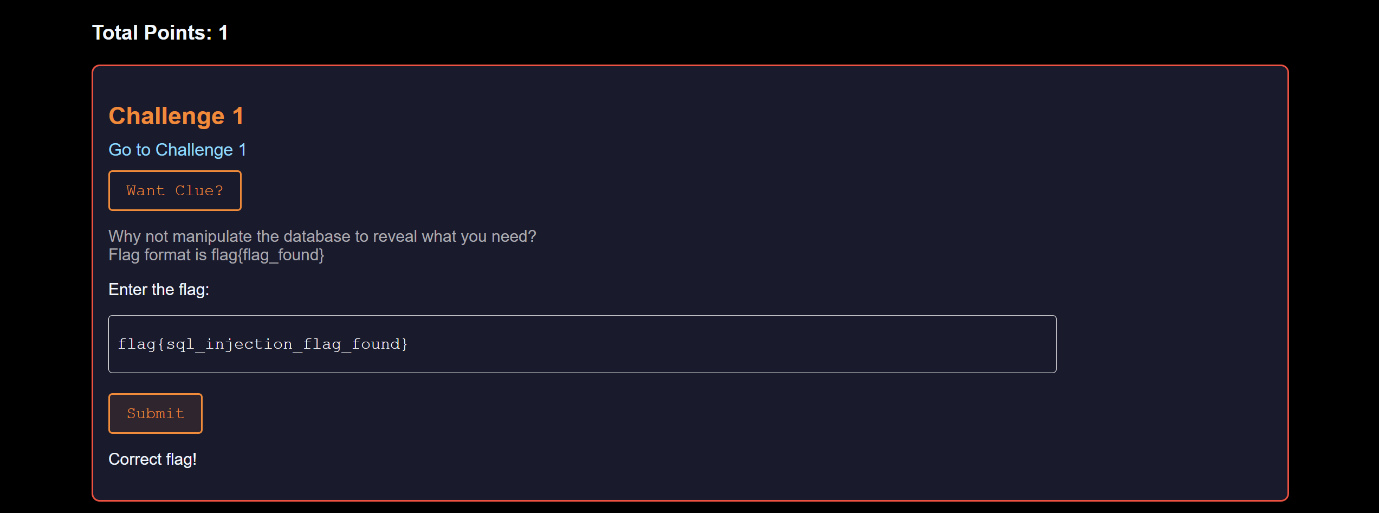
- Username: ' or 1=1 --

- Password: anything



3. **Submit the form to bypass authentication:**



****

**Conclusion**

By using SQL Injection, participants can bypass authentication and access the application.

**Challenge 2: Cross-Site Scripting (XSS)**

**Description**

This challenge involves exploiting an XSS vulnerability to execute a script and fetch sensitive data.

**Clue**

"Why not fetch more than just fruits? Maybe something you really want."

**Code and Vulnerability**

The vulnerability is in the way user input is displayed on the page:

**python**

# Vulnerable code in app.py

search\_query = request.form['search']

conn = get\_db()

cursor = conn.cursor()

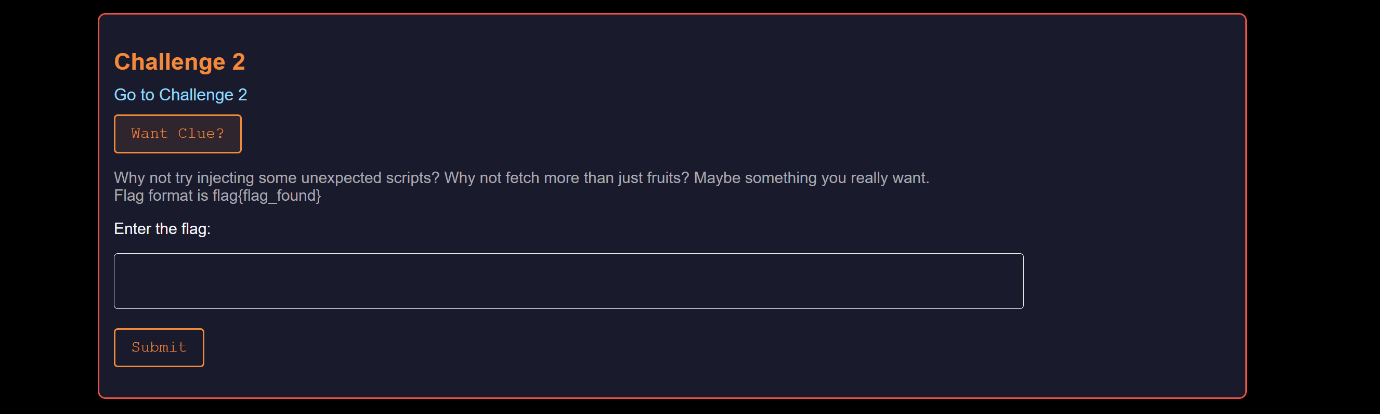
cursor.execute("SELECT \* FROM fruits WHERE name LIKE ?", ('%' + search\_query + '%',))

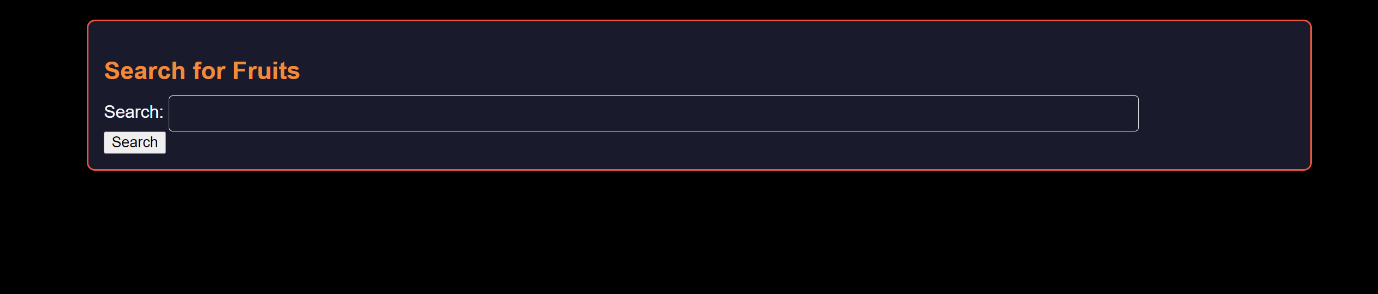
results = cursor.fetchall()

User input is directly included in the page without sanitization.

**Steps to Solve**

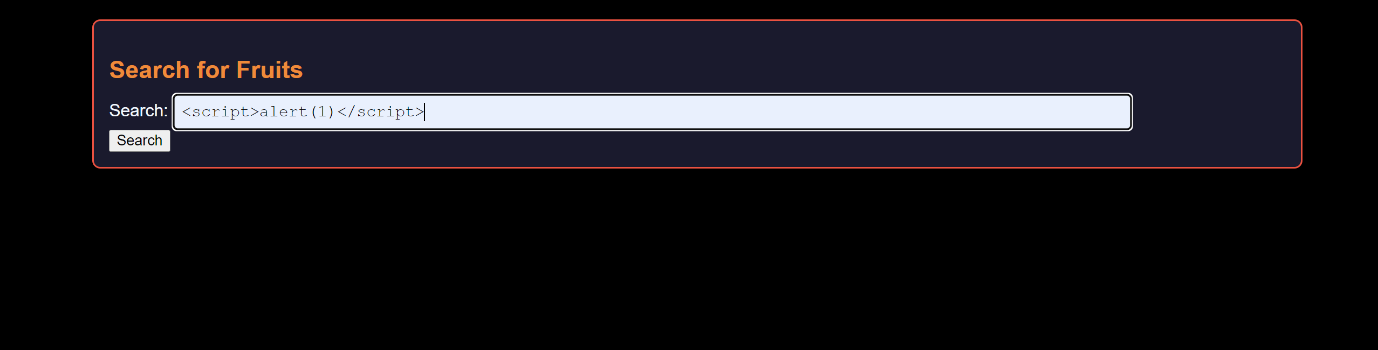
1. **Navigate to the form page:**

****

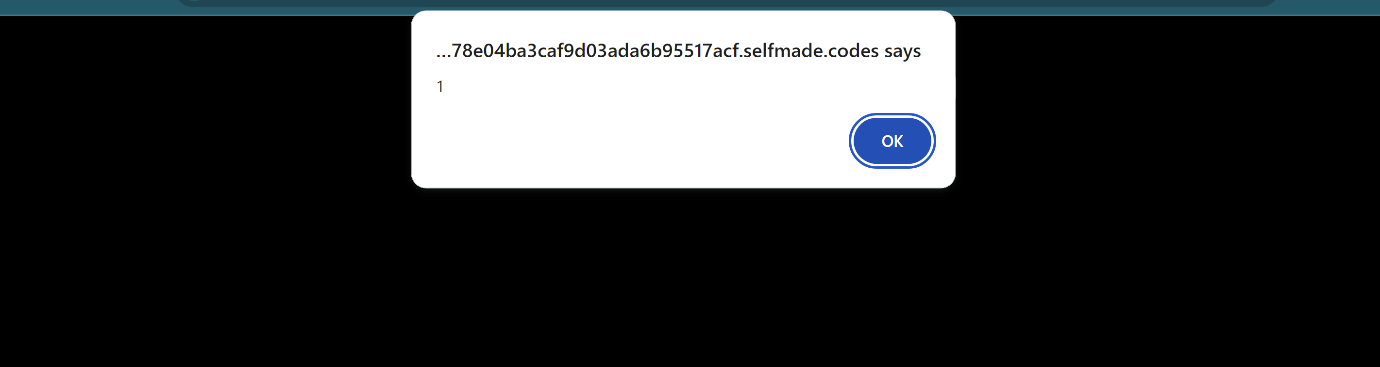
****

2**. Enter XSS Payload:**

Input: <script>alert('XSS');</script>

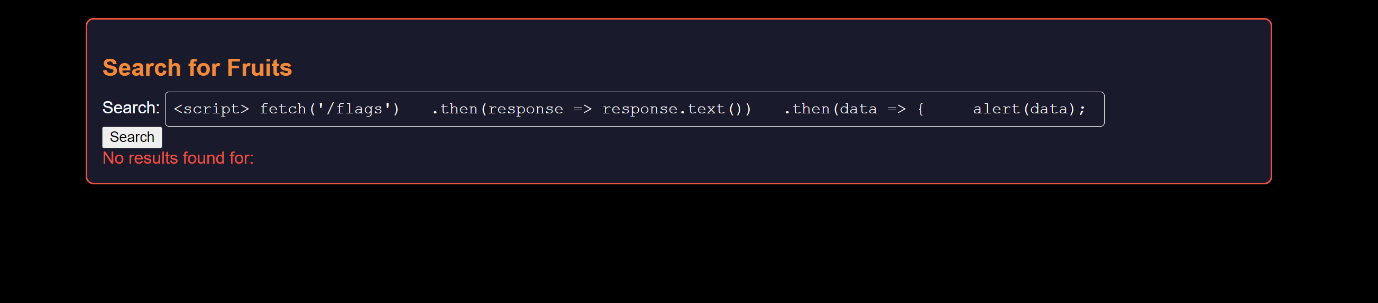
****

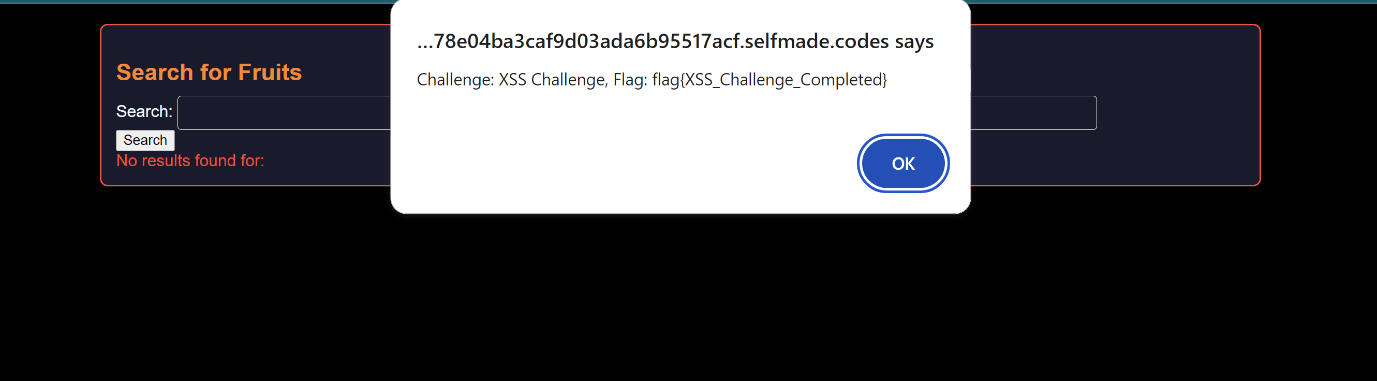
3**. Submit the form to trigger the alert:**

****

4. **Fetch Sensitive Data:**

Input: <script>fetch('/uploads/flags.txt').then(response => response.text()).then(data => alert(data));</script>

****

****

5. **Submit the form to retrieve the flag:**

****

**Conclusion**

By exploiting the XSS vulnerability, participants can execute arbitrary scripts in the browser and retrieve sensitive data, such as flags.

**Challenge 3: Command Injection**

**Description**

This challenge involves exploiting a command injection vulnerability to execute system commands.

**Clue**

"Can you get more than just a status response?"

**Code and Vulnerability**

The vulnerability is in the way system commands are constructed:

**python**

# Vulnerable part: Executing the user-provided `other\_command` directly

if other\_command.startswith('ls'):

output += '\n\n'

output += subprocess.check\_output('ls iot\_device\_files', shell=True).decode('utf-8')

elif other\_command.startswith('cat'):

parts = other\_command.split()

if len(parts) == 2:

filename = parts[1]

filepath = os.path.join('iot\_device\_files', filename)

if os.path.isfile(filepath):

output += '\n\n'

output += subprocess.check\_output(f'cat {filepath}', shell=True).decode('utf-8')

else:

output = 'File not found.'

else:

output = 'Invalid cat command. Usage: cat filename'

else:

output += '\n\n'

output += subprocess.check\_output(f'{other\_command}', shell=True).decode('utf-8') # Vulnerable

else:

output = 'Invalid command. Usage: status device\_name; ls/cat command'

else:

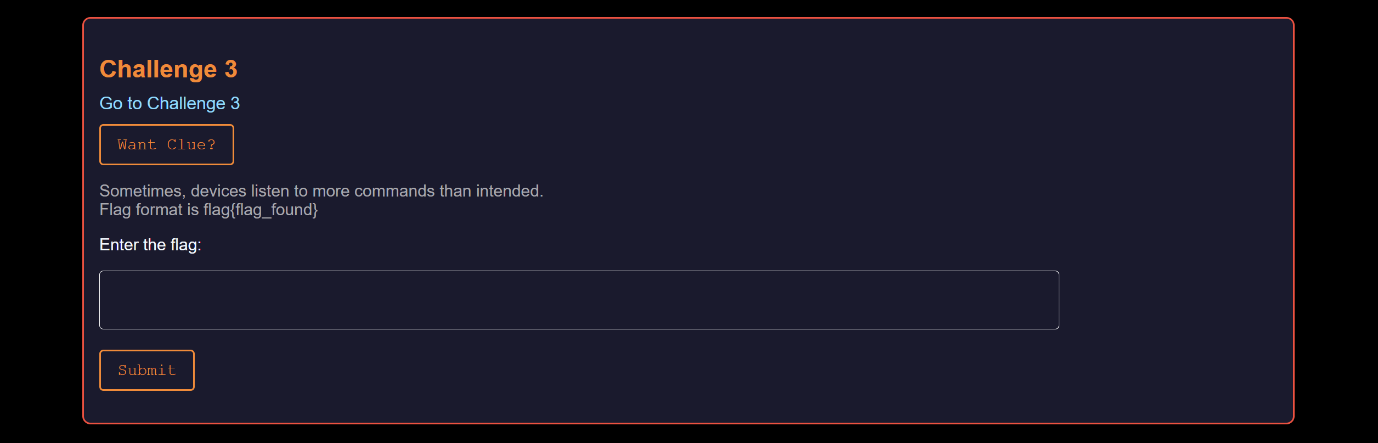
output = 'Invalid command. You must start with "status".'

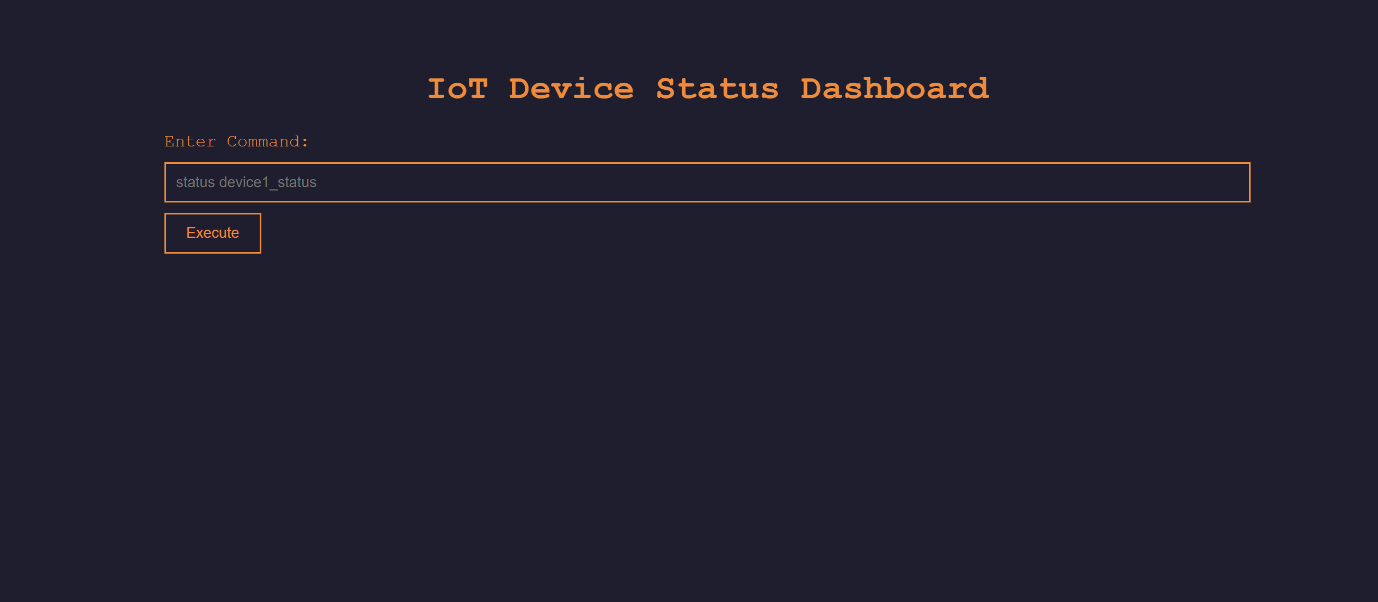
return render\_template('index.html', output=output)

User input is included in a system command without validation.

**Steps to Solve**

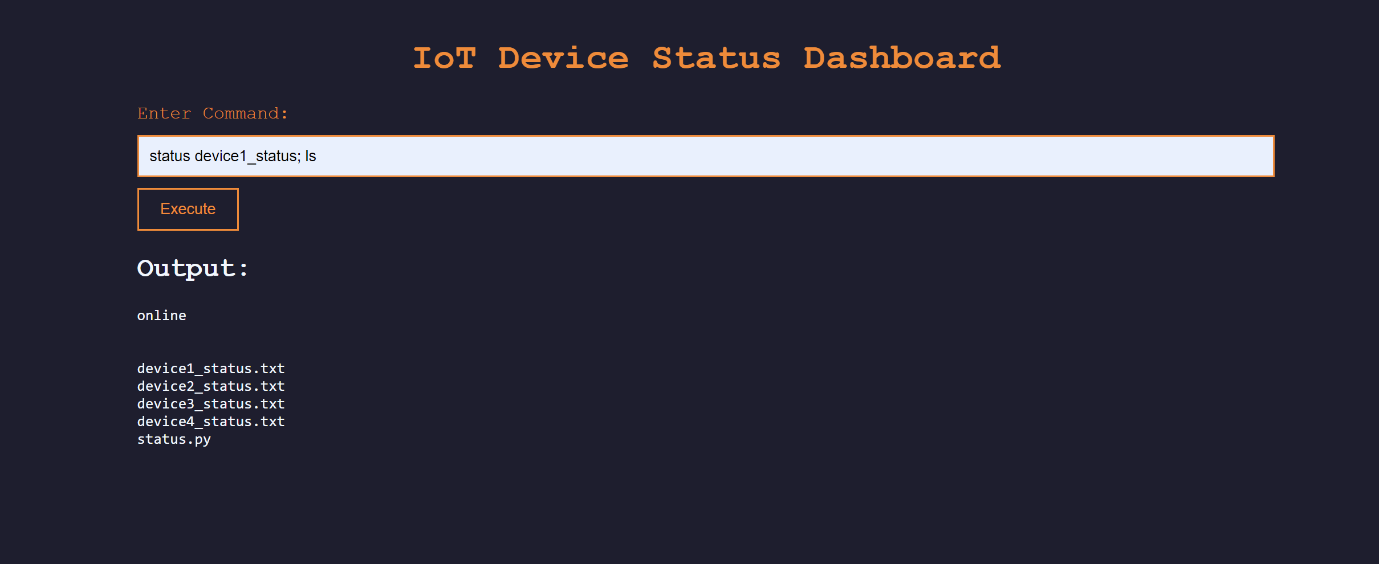
1. **Navigate to the status check page:**





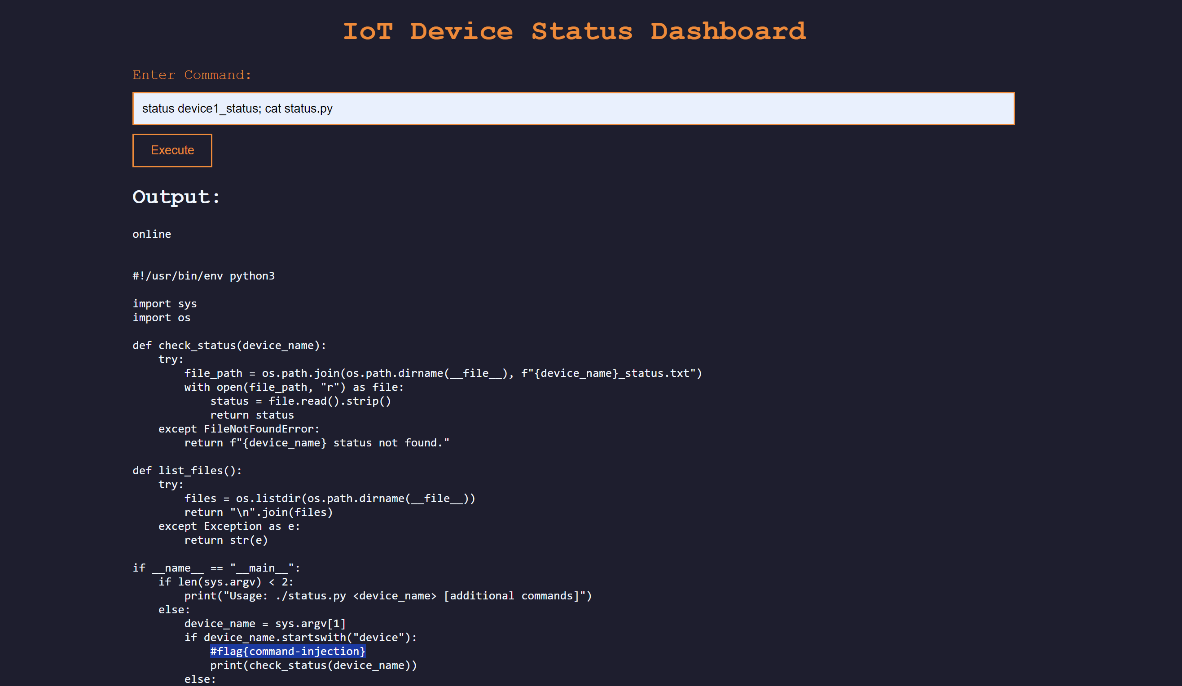
2. **Enter Command Injection Payload:**

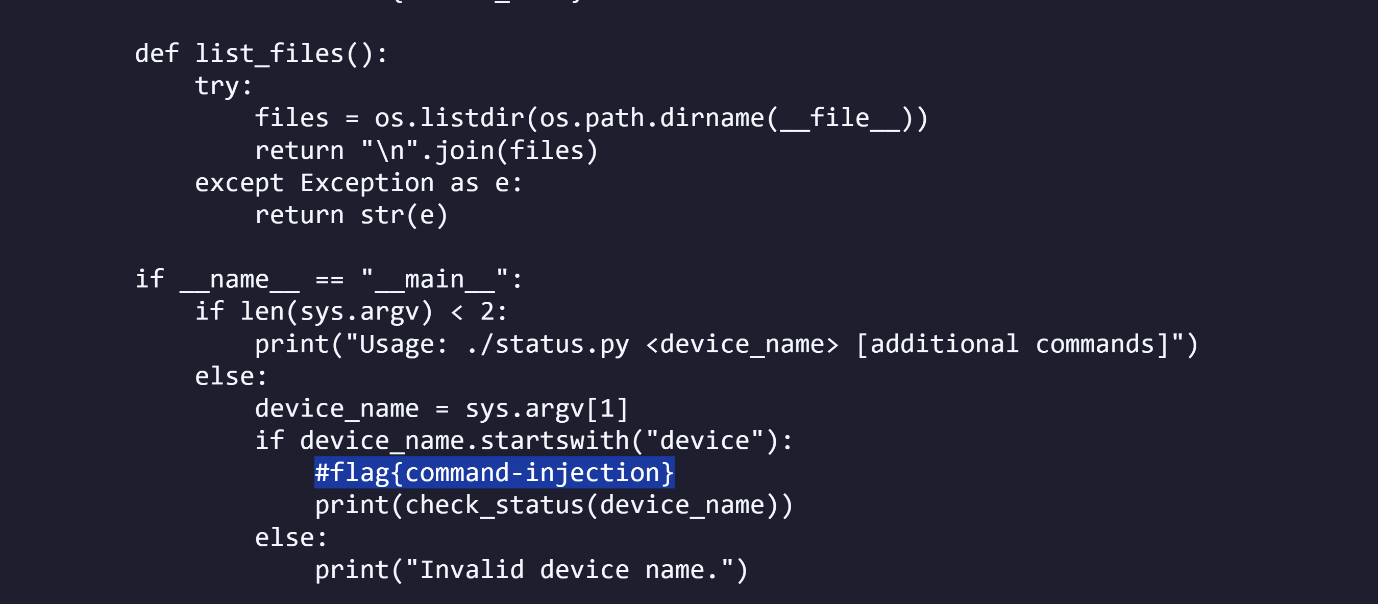
Input: device1\_status ; ls



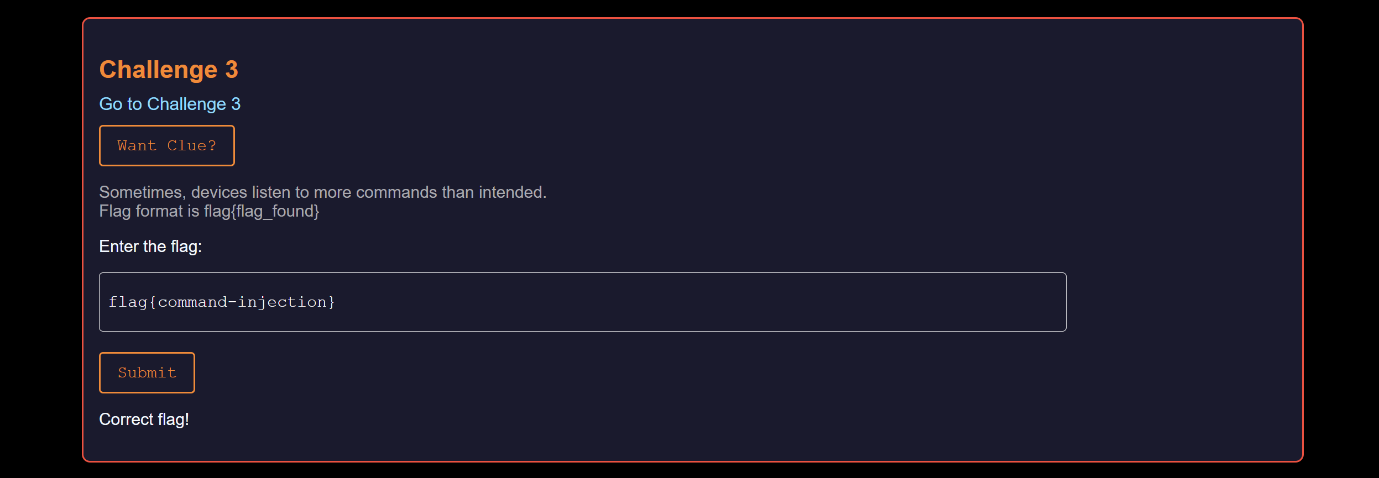
4. **Check Contents of a Specific File:**

Input: device1\_status ; cat status.py





5. **Submit the form to retrieve the flag:**



**Conclusion**

By exploiting the command injection vulnerability, participants can execute arbitrary commands on the server and retrieve sensitive data, such as flags hidden in specific files.

**Challenge 4: Insecure File Upload**

**Description**

This challenge involves exploiting an insecure file upload vulnerability to upload a malicious file and retrieve sensitive data.

**Clue**

"Can you upload more than just typical files?"

**Code and Vulnerability**

The vulnerability is in the way files are saved:

**python**

# Vulnerable code in app.py

if request.method == 'POST':

if 'file' not in request.files:

return 'No file part'

file = request.files['file']

if file.filename == '':

return 'No selected file'

allowed\_extensions = {'.sh', '.txt', '.jpg', '.png'}

if file and any(file.filename.endswith(ext) for ext in allowed\_extensions):

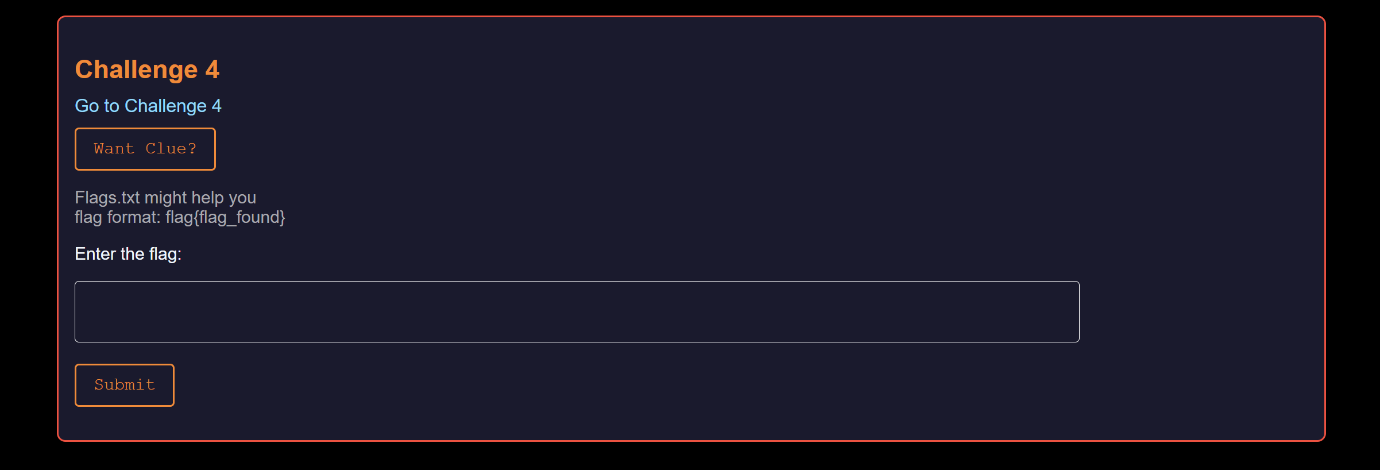
filepath = os.path.join('uploads', file.filename)

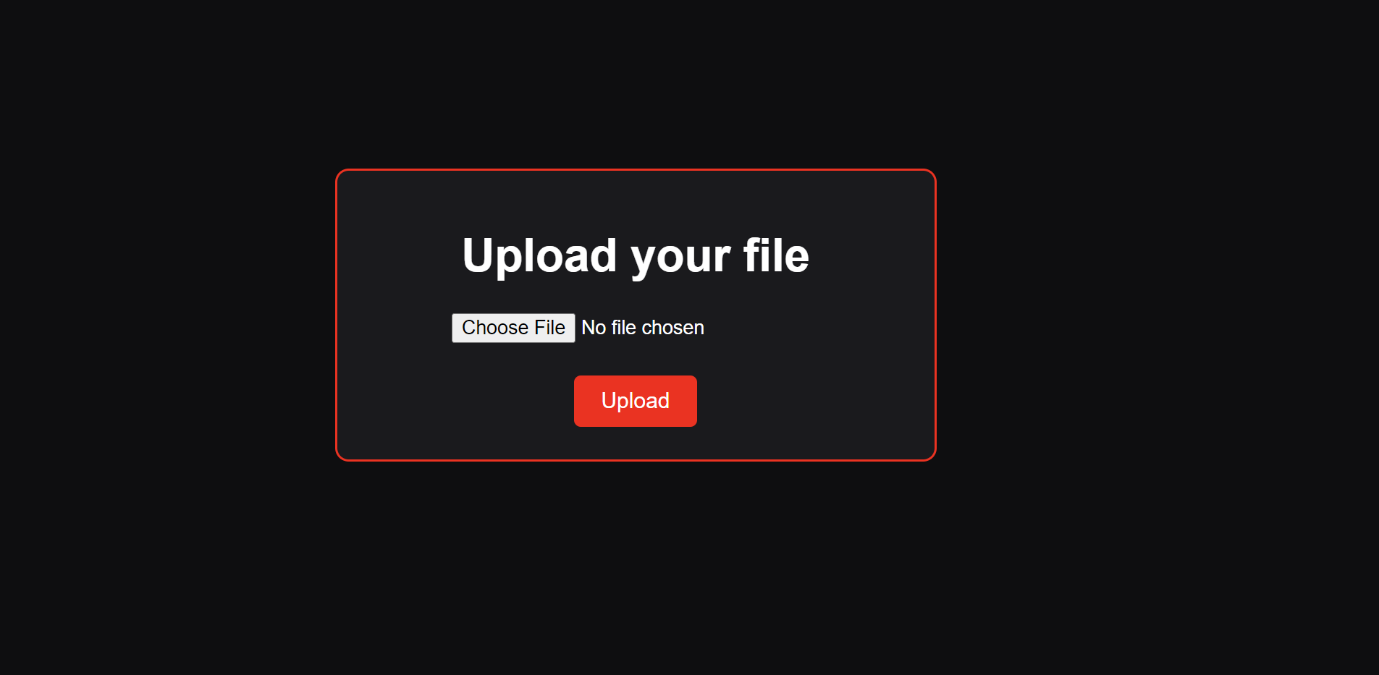
file.save(filepath)

The application does not properly validate or sanitize uploaded files.

**Steps to Solve**

1. **Navigate to the file upload page:**





2. **Upload Malicious File:**

Create a file exploit.sh with the following content:

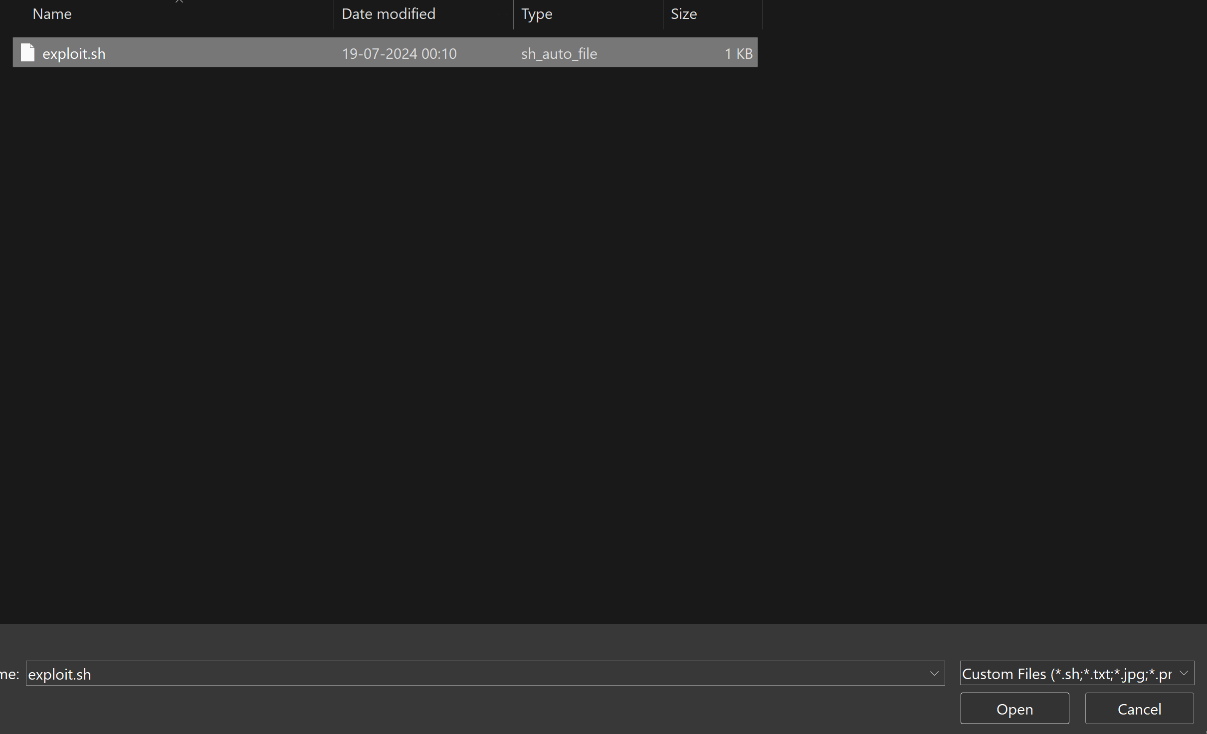
**exploit.sh**

#!/bin/bash

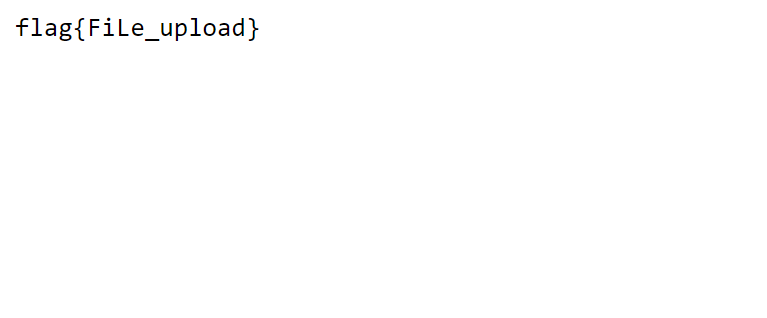
# This script will read the flags.txt file as a clue they have mentiones and output its contents

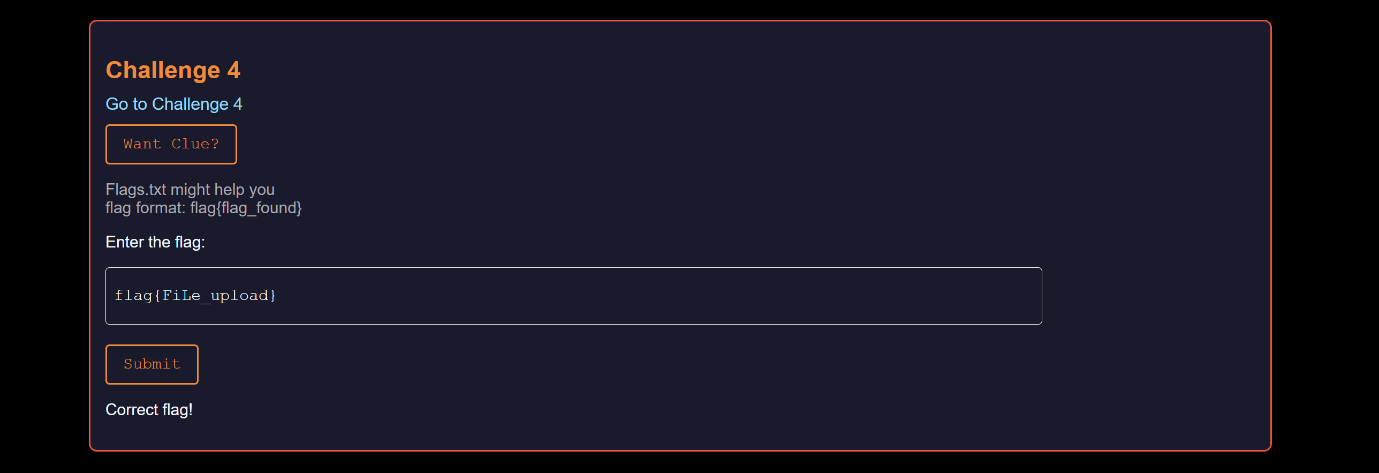
cat flags.txt

**Upload the file exploit.sh**

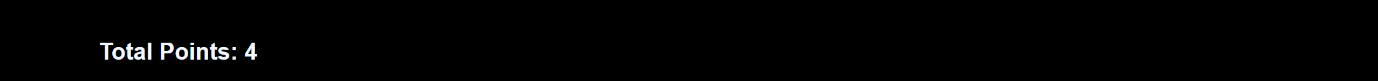


3. **Access the uploaded file to execute it:**





At the end finding all flags, we scored 4 points. Each for one challenge

****

**Conclusion**

By exploiting the insecure file upload vulnerability, participants can upload and execute malicious files on the server, retrieving sensitive data such as flags.

**Overall Conclusion**

This CTF challenge website provides cybersecurity enthusiasts with an opportunity to experience and practice crucial vulnerabilities in a real, practical way. By solving these challenges, participants can improve their understanding of web vulnerabilities and develop their penetration testing skills.