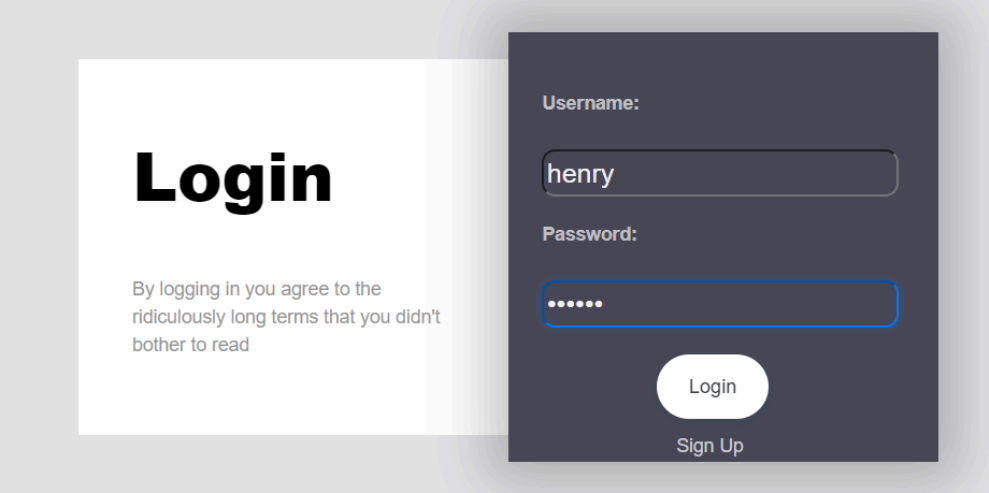
3 vulnerability

1. Login

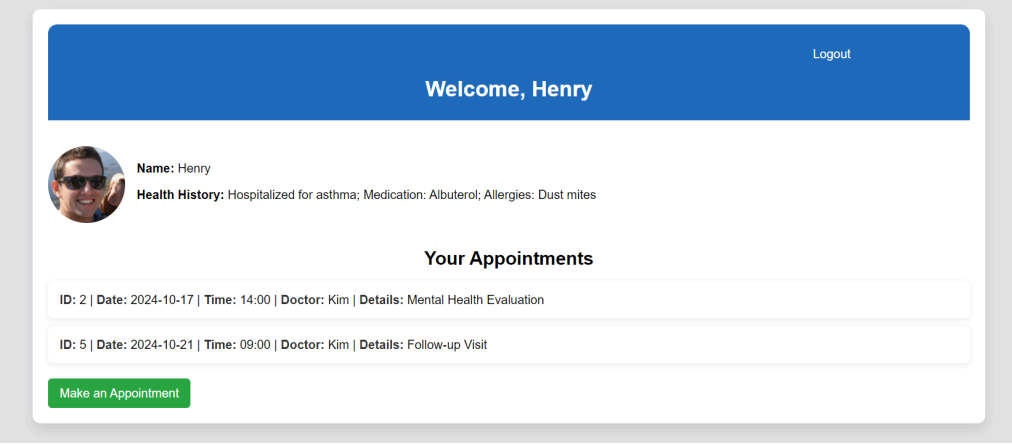
a.

	id	username	password	role
1	1	Kaylee	qwerty	Doctor
2	2	Kim	kim123	Doctor
3	3	Miya	password1	Patient
4	4	Weiss	password2	Patient
5	5	Michael	michael123	Patient
6	6	Henry	henry1	Patient

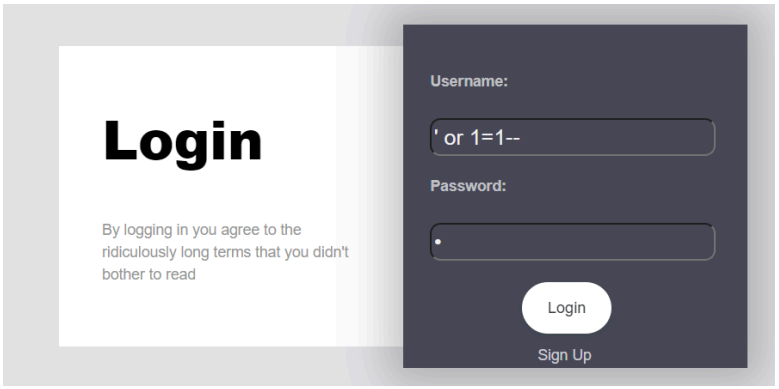
b.



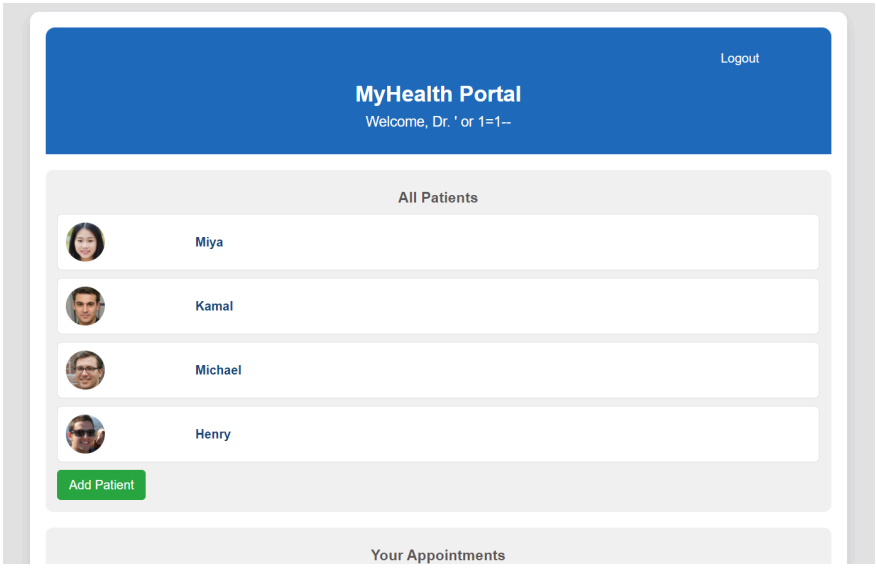
c.



2. SQL Injection

a. 

#password field -> anykey

b. 

#access portal as a dr

i. Code

```
username = request.form['username']
password = request.form['password']

# Vulnerable SQL query using string concatenation
query = f"SELECT * FROM users WHERE username = '{username.capitalize()}' AND password = '{password}'"

conn = sqlite3.connect('healthcare.db')
cursor = conn.cursor()

cursor.execute(query)
user = cursor.fetchone()

conn.close()
```

ii. How to mitigate

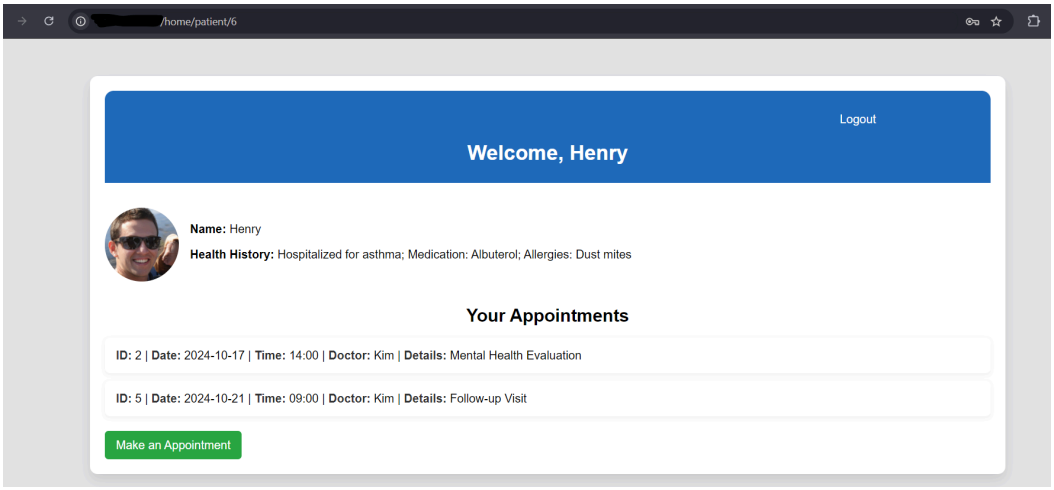
iii.

```python
if request.method == 'POST':
    username = request.form['username'].capitalize()
    password = request.form['password']

    # Use parameterized query to prevent SQL injection
    query = "SELECT * FROM users WHERE username = ? AND password = ?"

    conn = sqlite3.connect('healthcare.db')
    cursor = conn.cursor()


    cursor.execute(query, (username, password))
    user = cursor.fetchone()

    conn.close()
```

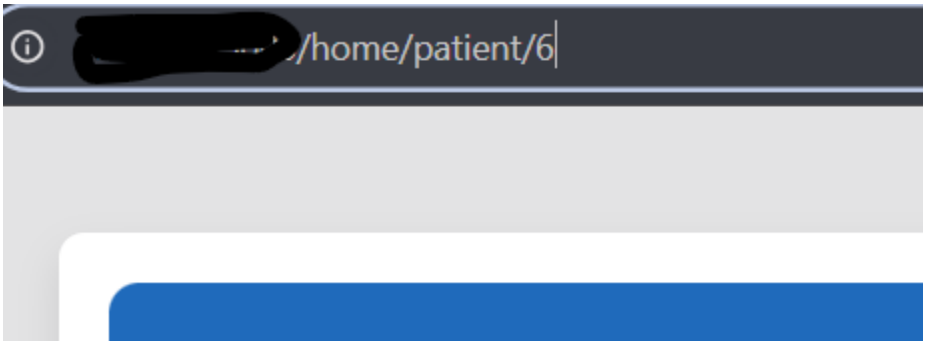
Invalid credentials

iv.

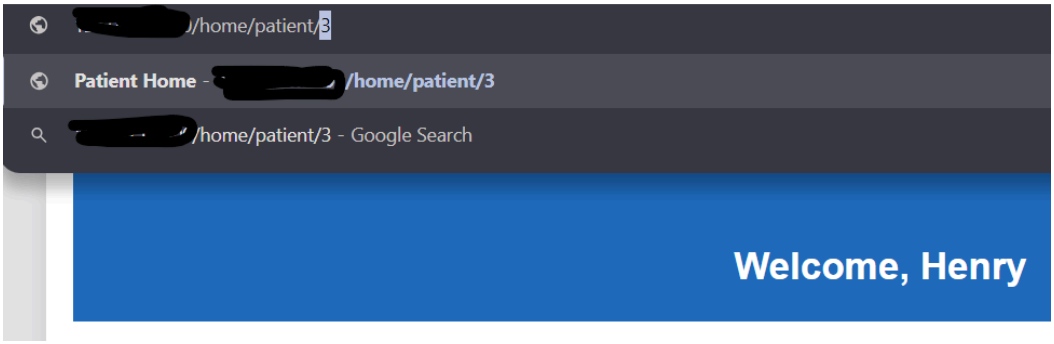
3. Insecure direct object references (IDOR)

a. 

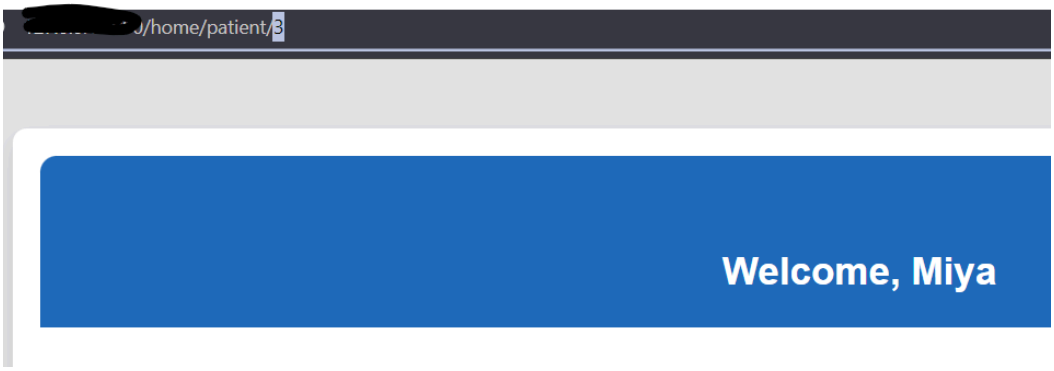
henry id is 6

b. 

c. Change henry id into 3

d. 

e. Accessing other patient data

f. 

g. 

h. Change henry role from patient to dr and accessing dr portal

i. 

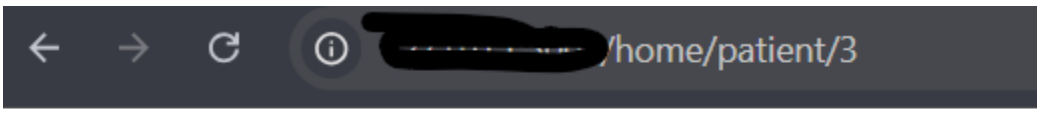
i. Code (this doesnt have any restriction)

ii.
```
if role == 'Patient':
    cursor.execute('SELECT patient_id FROM patients WHERE name = ?', (session['username'].replace("_", " "),))

# Find the patient by ID
cursor.execute('SELECT * FROM patients WHERE patient_id = ?', (patient_id,))
patient = cursor.fetchone()
```

iii. Mitigate (will filter user based on their role & id)

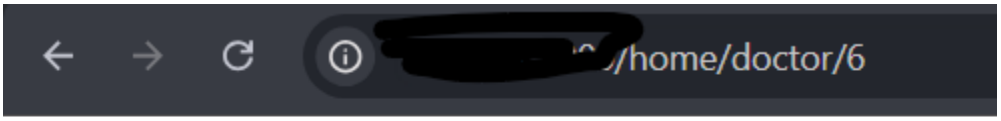
iv.

```
role = session.get('role')

# Ensure that only patients can access this route
if role != 'Patient':
    return "Unauthorized role for this page", 403

# Connect to the SQLite database
conn = sqlite3.connect('healthcare.db')
cursor = conn.cursor()

# Patients can only access their own record
cursor.execute('SELECT patient_id FROM patients WHERE name = ?', (session['username'].replace("_", " "),))
result = cursor.fetchone()

if result:
    own_patient_id = result[0]
else:
    return "Unauthorized", 403

# Ensure the patient can only access their own data
if own_patient_id != patient_id:
    return "Unauthorized access to another patient's data", 403

# Find the patient by ID
cursor.execute('SELECT * FROM patients WHERE patient_id = ?', (patient_id,))
patient = cursor.fetchone()

if not patient:
    return "Error patient not found", 404
```



Unauthorized access to another patient's data

j.

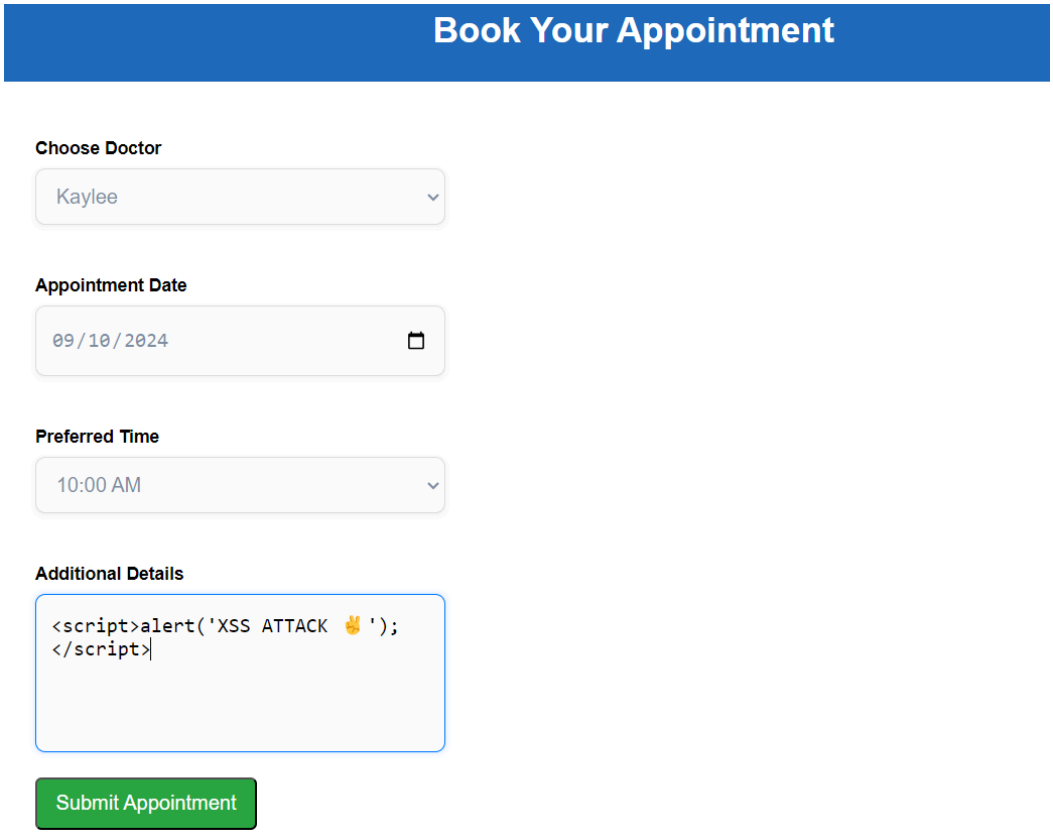


Unauthorized role for this page

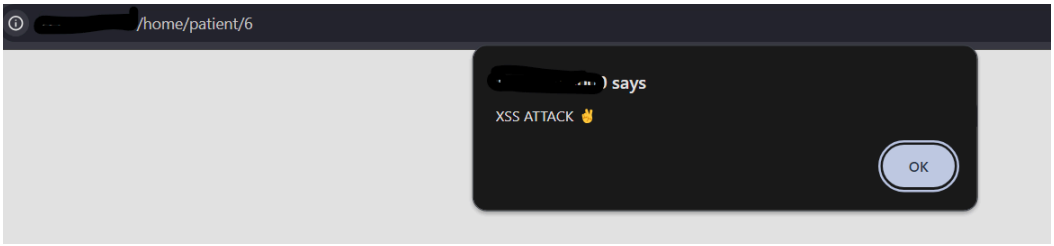
k.

4. Cross-site scripting (XSS)

a.



b.



i. Code (happen when the code not sanitize properly)

ii.

```
if request.method == 'POST':
    if role == 'Patient':
        # Find patient_id based on username
        cursor.execute('SELECT patient_id FROM patients WHERE name = ?', (session['username'].replace("_", " "),))
        patient_id = cursor.fetchone()
        if patient_id:
            patient_id = patient_id[0]
        else:
            return "Patient record not found", 404
    else:
        patient_id = request.form['patient_id']

    date = request.form['date']
    time = request.form['time']
    details = request.form['details']
    doctor_username = request.form['doctor']
```

iii. Mitigation (use bleach lib to remove any unwanted tag and script)

iv.

```
if request.method == 'POST':
    if role == 'Patient':
        # Find patient_id based on username
        cursor.execute('SELECT patient_id FROM patients WHERE name = ?', (session['username'].replace("_", " "),))
        patient_id = cursor.fetchone()
        if patient_id:
            patient_id = patient_id[0]
        else:
            return "Patient record not found", 404
    else:
        patient_id = request.form['patient_id']

    date = request.form['date']
    time = request.form['time']
    details = bleach.request.form['details'] ## bleach data input by removing unwanted tags and scripts
    doctor_username = request.form['doctor']
```

v.

ID: 6 | **Patient:** | **Date:** 2024-10-09 | **Time:** 10:00 | **Details:**

ID: 7 | **Patient:** | **Date:** 2024-10-10 | **Time:** 09:00 | **Details:** <script>alert('XSS ATTACK ✌ ');</script>

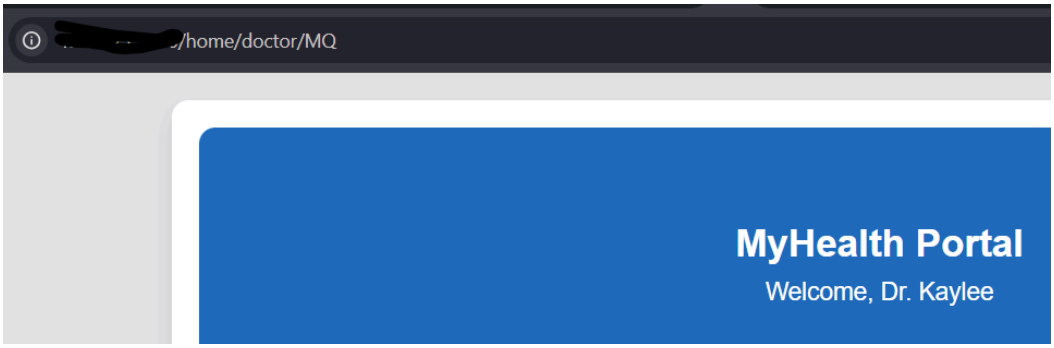
vi. Details for id 6 is null due to the xss script not be detect as string while id 7 the script is been detected as string because of santilize data using bleach

5. Extra security
   a. Obfuscating the URL using Base64 encoding

b.

```
# obfuscate id using based64 encoding and decoding
def obfuscate(id):
    # Convert id to a string, encode to bytes, then base64 encode
    encoded_bytes = base64.urlsafe_b64encode(str(id).encode('utf-8'))
    # Strip the padding "=" for a cleaner URL
    return encoded_bytes.decode('utf-8').rstrip('=')

def deobfuscate(obfuscated_id):
    padding = '=' * (4 - (len(obfuscated_id) % 4))
    obfuscated_id += padding
    # Base64 decode the obfuscated ID back to original id
    decoded_bytes = base64.urlsafe_b64decode(obfuscated_id.encode('utf-8'))
    return int(decoded_bytes.decode('utf-8'))
```

c.



   d. Flask “SECRET_KEY” (Create a unique secrete_key to prevent session haijacking)

e.

```python
# Generate a random secret key if not set in environment variable
if 'SECRET_KEY' not in os.environ:
    secret_key = secrets.token_hex(16)  # Generates a random key
else:
    secret_key = os.environ['SECRET_KEY']  # Load from environment variable

app = Flask(__name__)
app.secret_key = secret_key  # Used for session management
```

f. Content Security Policy (CSP) - further mitigate xss

g.

```python
# Add Content Security Policy (CSP)
@app.after_request
def add_security_headers(response):
    response.headers['Content-Security-Policy'] = "default-src 'self'; script-src 'self';"
    return response
```