SQL Injection 101

Batman's Kitchen 2024 | Workshop 1

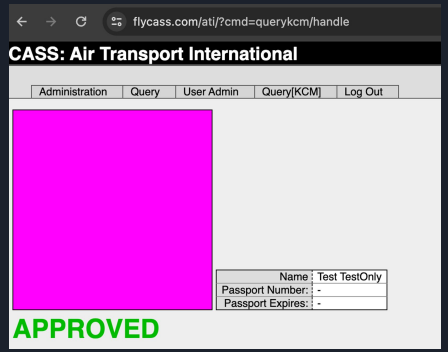
General Announcements

Next week’s meeting is a talk about Burp suite and web hacking

- Beginner-friendly workshop
- Wednesday the 9th, 5:30 PM - 7:30 PM

Recently in the news...

- Ever wanted to bypass airport TSA?
- Use SQL Injection! (https://ian.sh/tsa)
  - A basic SQL Injection allowed editing a TSA database to approve random people to bypass TSA lines



SQL Basics

What is SQL?

- Stands for “Structured Query Language”
- Programming language which allows you to interact with databases
- A database consists of tables which contain columns. Think of a table in Excel, for example.
- Comes in many different flavors (SQLite vs MySQL vs Postgre)

SQL Query Structure

SELECT username, password **FROM** credentials **WHERE** username = “user” **AND** password = “pass”;


Keywords


Table Name


Column Name

Other Important SQL Keywords

UNION - Combines the result of 2+ SELECT statements

LIKE - Looks for a specified pattern (ie LIKE “hi” selects “hire”)

DROP - Removes from the table

OR/NOT/AND - Logical Operands

ORDER BY x (ASC/DESC) - Sorts a query by a specific row name

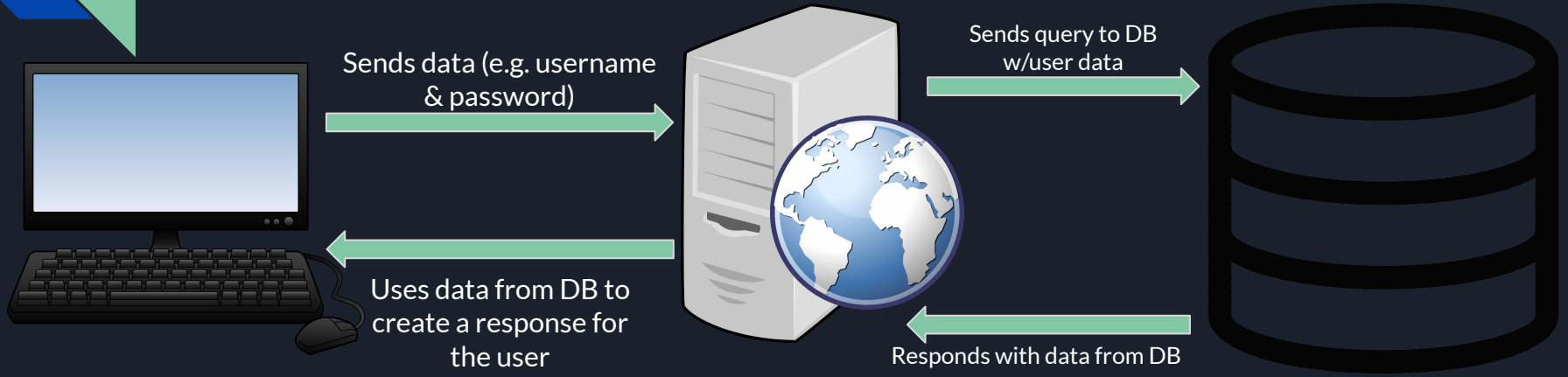
LIMIT x - Limits the query to x values

SQL Syntax notes

- Data types must match when selecting
  - Can’t select a string value from int field
- Number/field type of queries must match when union selecting
  - If a select selects int, string, string, the other must as well
- Different variations of SQL can have different rules

SQL Injection

How Websites (often) use SQL



Web browser (AKA Client / User)

Server

Database (DB)

Example: Login Page

- User submits their username and password to the server.
- Server inserts username and password into:
  - SELECT user FROM Users WHERE Username = ‘{username}’ AND Password = ‘{password}’;
- Database runs the query, responds with info about the user.
- Server uses that info to generate the user’s home page and send it to them.
- If the username/password are incorrect, the DB won’t find any users with that combination and the server will say that the login failed.

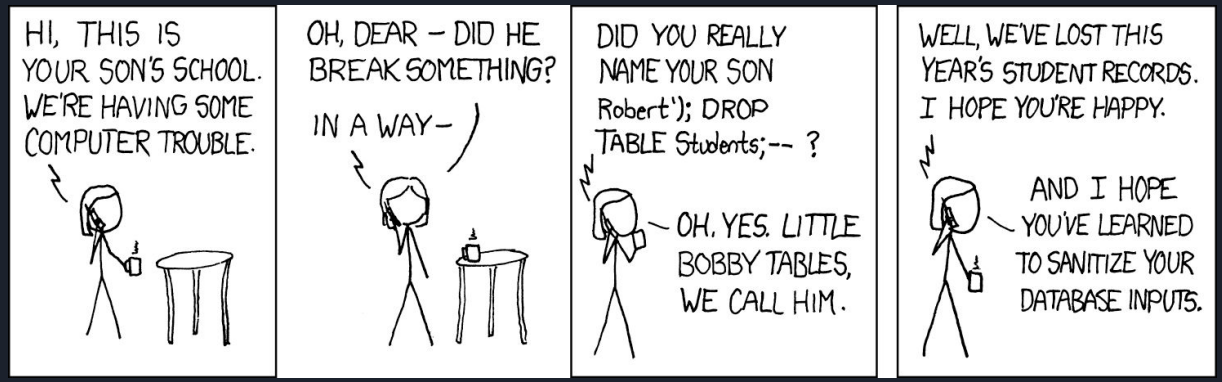
Evil hacker stuff (SECRET)

- What happens if a user puts in special SQL characters as part of their username? For example, what if they submit their username as admin';--
- SELECT user FROM Users WHERE Username = 'admin';-- ' AND Password = '';

```
SELECT user from Users WHERE Username = 'admin';-- ' AND Password = '';
```

Evil Hacker Stuff (Even More Secret)

- If you find a way to do SQL injection, you can generally run whatever SQL code you want.
- Some fun things you can do:
  - DROP TABLE
  - SLEEP(10000000)
    - Some versions of SQL don't have a SLEEP so this doesn't always work



https://xkcd.com/327/

Mitigations

- Sanitized Input/Keyword Blacklists
  - Remove/Disallow specific SQL keywords/structures when checking user queries
  - Won't always work, see challenges for why
- Prepared Statements **(USE THESE!!!)**
  - Create a query template with specified input parameters
  - Separates data from the query! Python example below:

```
import mysql.connector
with mysql.connector.connect(database="mysql", user="root") as conn:
    with conn.cursor(prepared=True) as cursor:
        # Assume that there's some user input here
        params = ("' OR '1' = '1')--", "hehe i SQLi now")
        cursor.execute("SELECT * FROM users WHERE username = %s AND password = %s", params)
        print(cursor.fetchall())
```

Fun Challenges

Challenge 1: Dump everything from one table

http://128.95.1.8:5000

```
@app.route("/challenge1", methods = ['POST'])
def challenge1():
    search = request.form["search"]

    try:
        results = conn1.cursor().execute(
            "SELECT * FROM challenge WHERE bread_name='" + search + "'"
        ).fetchall()
        return json.dumps(results)
    except sqlite3.Error as err:
        return json.dumps([['error:', str(err)]])
```

Challenge 2: There's a blocklist now

```python
def has_numbers(s):
    return any(char.isdigit() for char in s)

@app.route("/challenge2", methods=["POST"])
def challenge2():
    query = request.form["search"]

    # I heard blacklists are secure...
    if ('or' in query) or ('=' in query) or has_numbers(query):
        return json.dumps([["Hack detected!", "No flag for you..."]])

    try:
        results = conn2.cursor().execute(
            f"SELECT * FROM challenge WHERE bread_name='{query}'"
        ).fetchall()
        return json.dumps(results)
    except sqlite3.Error as err:
        return json.dumps([['error:', str(err)]])
```

Challenge 3: Blocklists 2

Just skip this one it turned out to be too hard because Daniel is built different

Challenge 4: Leak data from a different table

```
@app.route("/challenge4", methods=["POST"])
def challenge4():
    query = request.form["search"]

    try:
        results = conn4.cursor().execute(
            f"SELECT * FROM decoy WHERE c1='{query}'"
        ).fetchall()
        return json.dumps(results)
    except sqlite3.Error as err:
        return (json.dumps([['error:',str(err)]]))
```

Challenge 5: Blind SQL

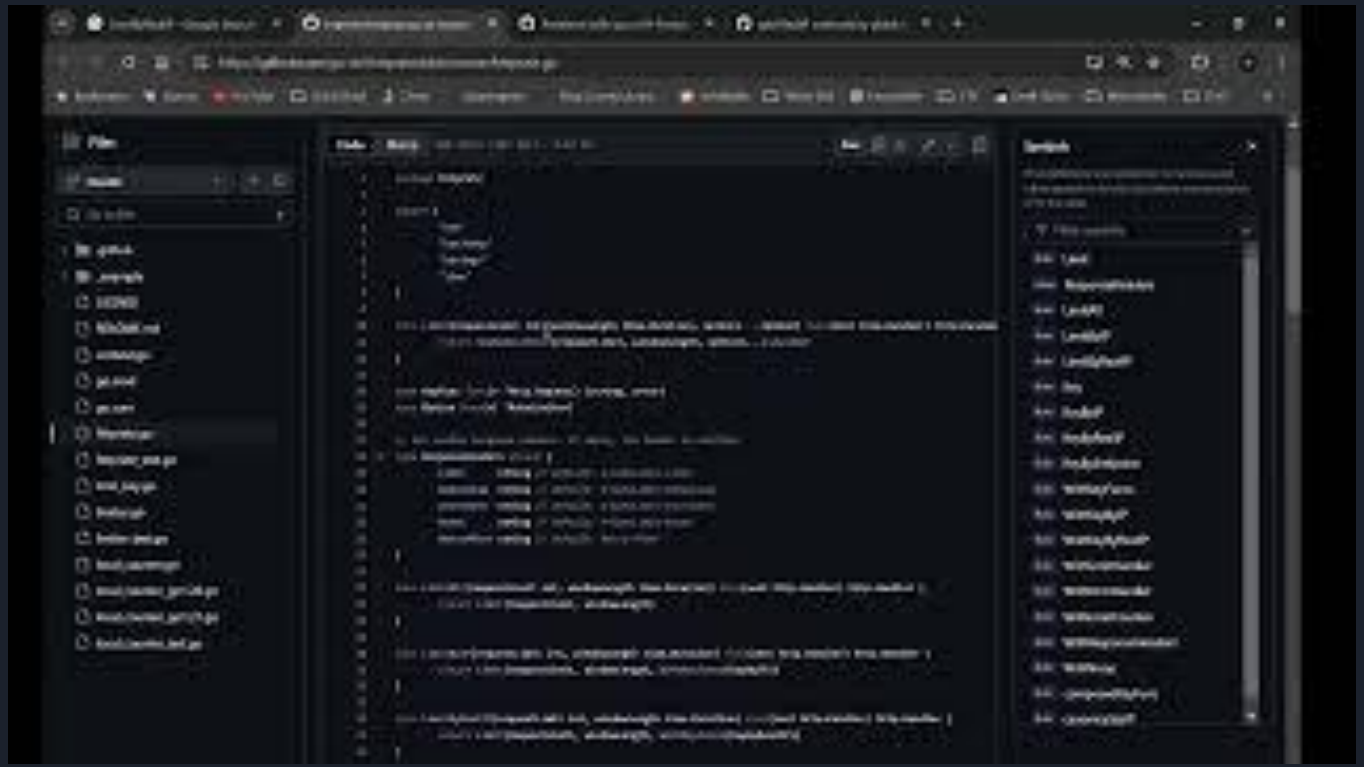
```python
@app.route("/challenge5", methods=["POST"])
def challenge5():
    query = request.form["search"]

    # Wait.. what if I just don't show you output anymore :D
    # The bakery is closed you can't hack me anymore.

    try:
        stmt = f"SELECT * FROM challenge WHERE bread_name LIKE '{query}'"
        print(stmt)
        results = conn5.cursor().execute(
            stmt
        ).fetchall()

        # I found this cool random function that gives you garbage output!
        # If I ask it for a lot of bytes it takes some time though
        # I wonder if that's a problem somehow..
        # fake_results = conn5.cursor().execute(
        #     f"SELECT randomblob(10)"
        # ).fetchall()

        output = [('no output for you!',)]

        return json.dumps(output)
    except sqlite3.Error as err:
        return (json.dumps([['error:',str(err)]]))
```

Additional Practice

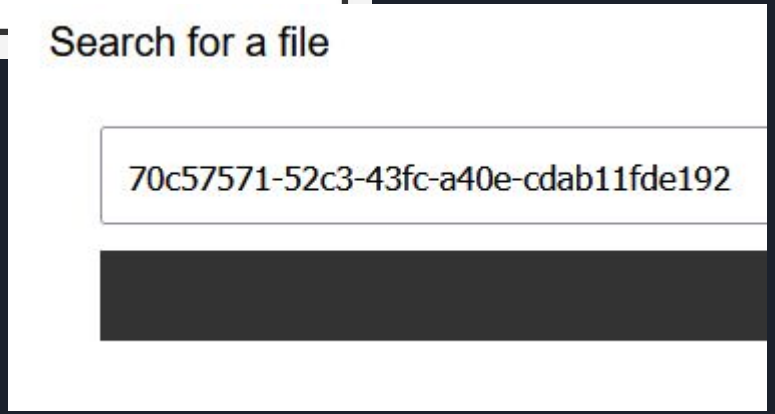
- Portswigger SQLi Labs: https://portswigger.net/web-security/all-labs#sql-injection
- picoGym (https://play.picoctf.org/practice) Challenges:
  - SQLiLite
  - MoreSQLi
  - SQL Direct

Buckeye Recap

Dojo - Presented by Jonathan



SSFS File Upload



The requests on the backend

```
127.0.0.1 - - [01/Oct/2024 10:57:10] "GET /static/css/style.css HTTP/1.1" 304 -
127.0.0.1 - - [01/Oct/2024 10:57:26] "POST /upload HTTP/1.1" 200 -
1
70c57571-52c3-43fc-a40e-cdab11fde192
127.0.0.1 - - [01/Oct/2024 10:57:34] "GET /search/70c57571-52c3-43fc-a40e-cdab11fde192 HTTP/1.1" 200 -
127.0.0.1 - - [01/Oct/2024 11:00:00] "GET /download/70c57571-52c3-43fc-a40e-cdab11fde192 HTTP/1.1" 200 -
```

ignore the print statements

images are uploaded via a POST request, and the search and downloads are GET requests formatted with /search/{path}

we want to get to the flag.txt file which is stored at /flag.txt on the server (and in the main folder when hosted locally)

Relative Paths

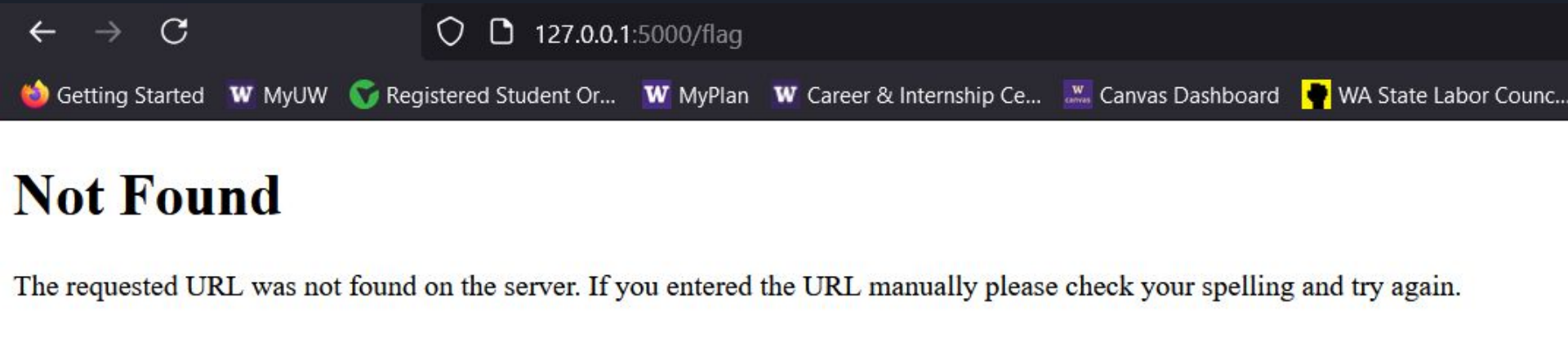
../ = previous/upper directory

When in /home/user/SSFS/uploads/

/flag = ../../../../../flag

So:

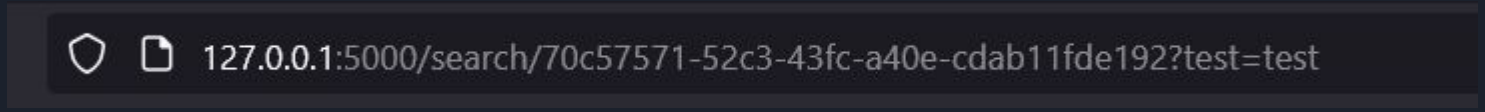
GET /downloads/../../../../flag ?

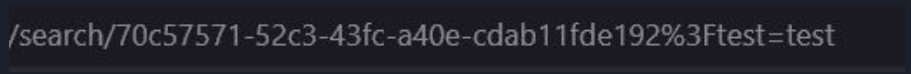


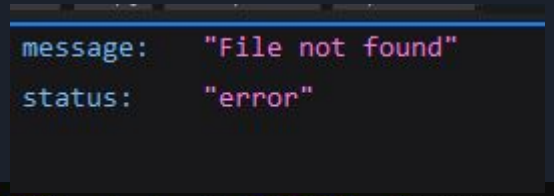
HTTP Requests are also paths so this the ../ is applied when routing the request

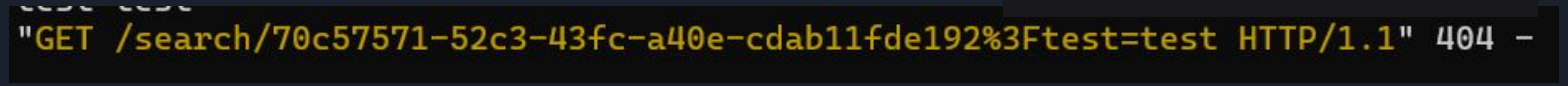
URL Encoding

URL Encoding is how you indicate parts of the URL should not be read as syntax









We can locate and download the flag with %2F