Тема 4. SQL-инъекции

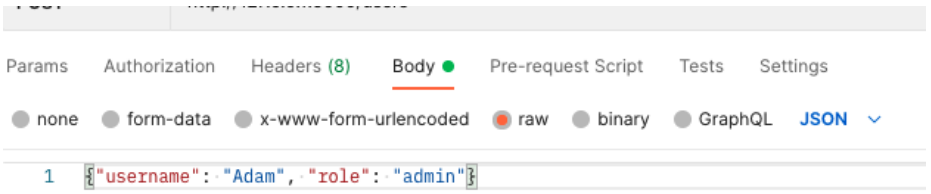
Напоследок поговорим об одной интересной уязвимости, с которой мы, сами того не подозревая, уже справились, — об SQL-инъекциях.

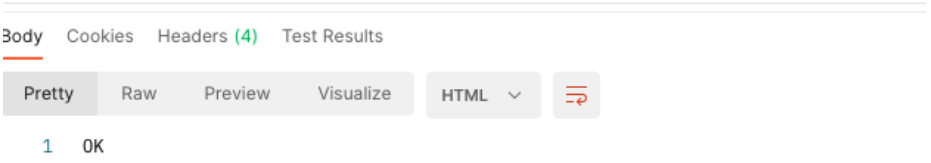
```
import json
import sqlite3

from flask import Flask, request, g

create_db_sql = """
DROP TABLE IF EXISTS `users`;

CREATE TABLE `users` (
   id INTEGER PRIMARY KEY AUTOINCREMENT,
   username VARCHAR(255) NOT NULL,
   role VARCHAR(255) NOT NULL
);
"""

app = Flask(__name__)

def get_db():
   db = getattr(g, '_database', None)
   if db is None:
       db = g._database = sqlite3.connect("db_users.db")
   return db


@app.teardown_appcontext
def close_connection(exception):
   db = getattr(g, '_database', None)
   if db is not None:
       db.close()


def create_db():
   with sqlite3.connect("db_users.db") as conn:
       cursor = conn.cursor()
       cursor.executescript(create_db_sql)


@app.route('/users')
def get_users():
   conn = get_db()
```

```
    cursor: sqlite3.Cursor = conn.cursor()
    cursor.execute("SELECT * FROM users")
    users = cursor.fetchall()
    return json.dumps(users)


@app.route('/users', methods=['POST'])
def create_user():
    data = request.get_json()
    conn = get_db()

    cursor: sqlite3.Cursor = conn.cursor()
    cursor.executescript(
        f"""
            INSERT INTO
                `users`(username, role)
            VALUES
                ("{data['username']}", "{data['role']}");
        """
    )
    conn.commit()
    return 'OK', 200


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

Напишем простой веб-сервер на Flask и со SQLite-библиотекой под капотом.

В сервисе две ручки, в одной мы получаем список пользователей, в другой добавляем этих пользователей в базу. Заметим, что в таблице у сущности «Юзер» есть специальное поле — его роль. Можем представить, что это поле выделяет аккаунт админа среди других и даёт ему дополнительные права на весь сервис. Попробуем добавить парочку юзеров и оставим пока за скобками некоторую ущербность бизнес-логики самого приложения. Сделаем парочку запросов, проверим, что всё работает.





Вроде бы да. Но вот, одним днём, некто очень коварный решил воспользоваться уязвимостью, которую мы по оплошности оставили, и отправляет на сервер такой json:

{"username": "Adam\", \"user\"); update `users` set role=\"admin\" where username = \"John\";", "role": "admin"}

Как видно, тут не совсем правильное имя, а, кажется, SQL-запрос, именно так — нам делают SQL-инъекцию наподобие того, как мы это делали с JS.

Если мы выполним этот скрипт, то увидим, что пользователь Джон внезапно возвысился в своей роли.

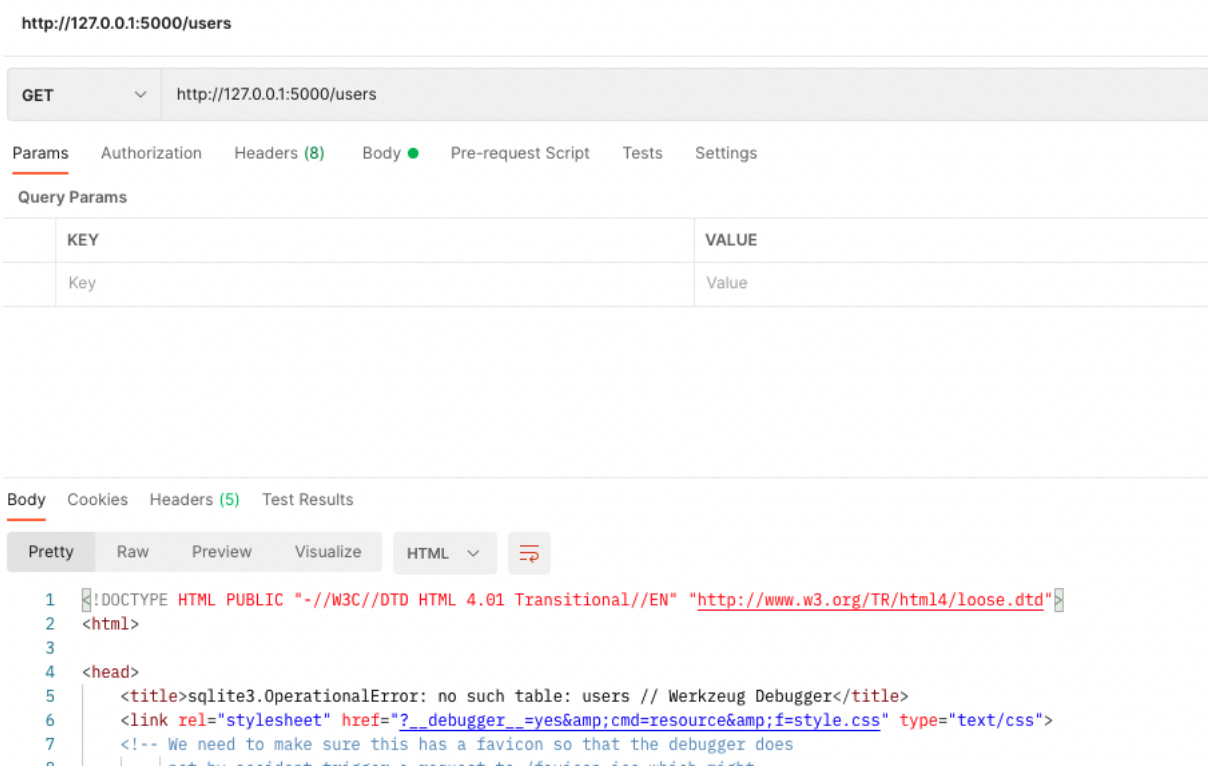
Да, операция упала где-то в конце, тем не менее мы добились своего.

Или ещё один пример, ставший притчей во языцех:

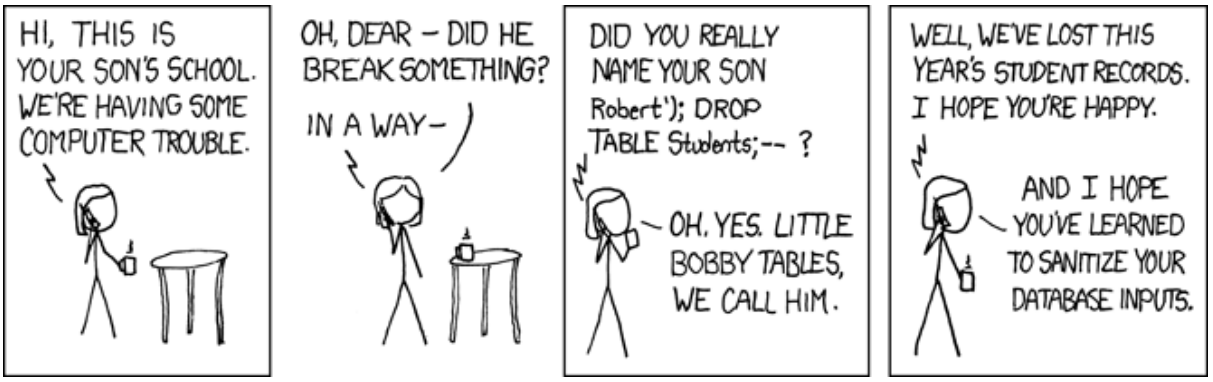
{"username": "Adam\", \"user\"); drop table `users`;", "role": "admin"}

Просто удаляем таблицу.

Если сделаем GET-запрос, то увидим, что таблицы больше нет:



Теперь вы, скорее всего, поймёте смысл одной известной картинки:



Как мы уже этого избегали? Прежде всего мы, как правило, пользовались командой execute для вставки в таблицу. А эта команда не даст выполнить более одного выражения зараз. Далее мы никогда не подставляли данные напрямую в запрос через форматирование строки, а передавали их как аргументы метода. А уже внутри метода SQL-драйвер делал все необходимые проверки, чтобы защитить нас от инъекций.

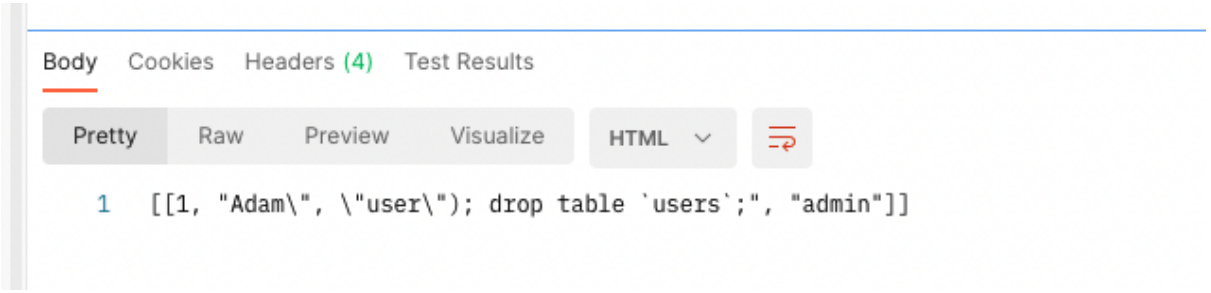
Проверим, как это сработает?

```
cursor.execute(
    f"""
        INSERT INTO
```

```
            `users`(username, role)
        VALUES
            (?, ?);
    """,
    (data['username'], data['role'])
)
```

Перепишем код по добавлению записи в таблицу в безопасный вид и вызовем эндпоинт.

Проверим, что лежит в базе:



Видим, что страшная инъекция превратилась в простую строку.

Итак, в этом модуле мы познакомились с основами безопасности веб-приложений и узнали, какие типы атак наиболее распространены. Знаем, что такое CORS, XSS и CSRF. Впереди немного практики, до встречи в следующих модулях!