xbrutovskyf
FIIT STU

# Protection against SQL injection attacks

xbrutovskyf
FIIT STU

# 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.

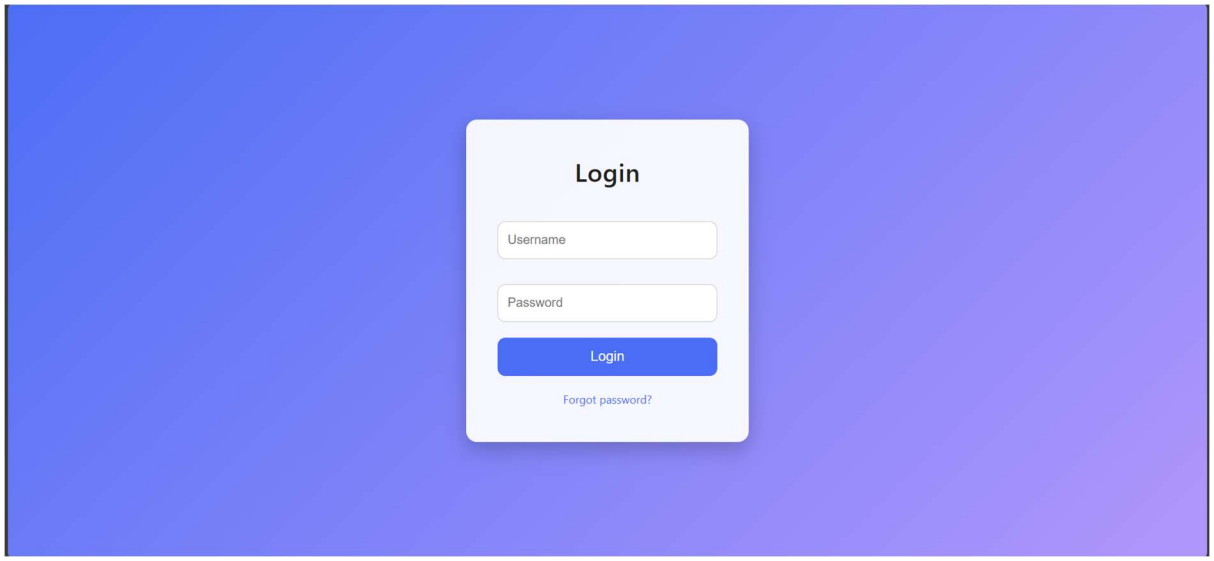



*Figure 1 Login page in vulnerable web application*

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.

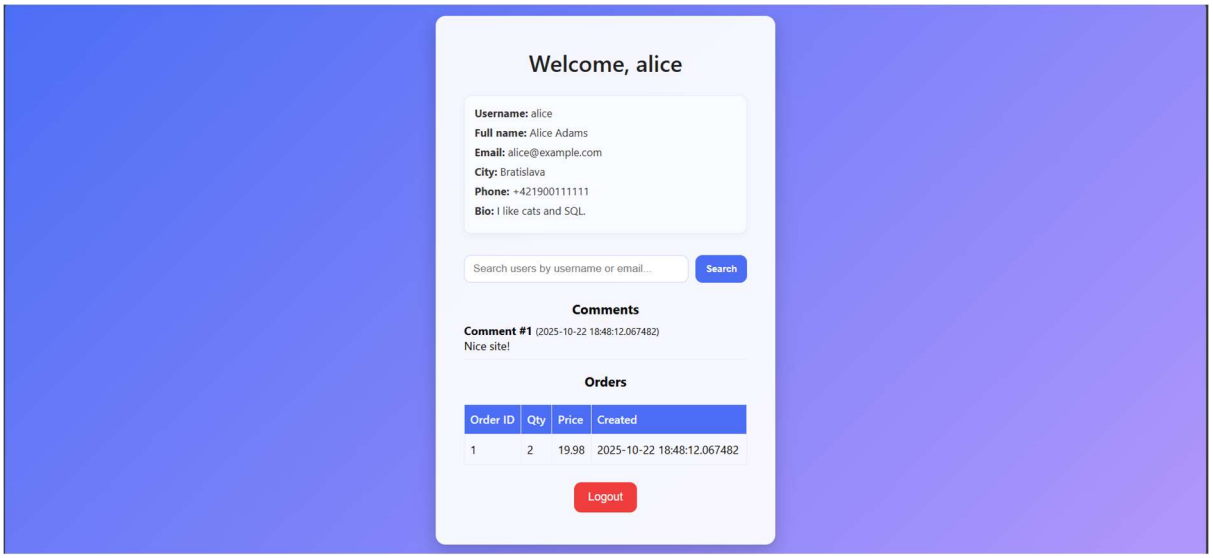



*Figure 2 Dashboard page in vulnerable web application*

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.

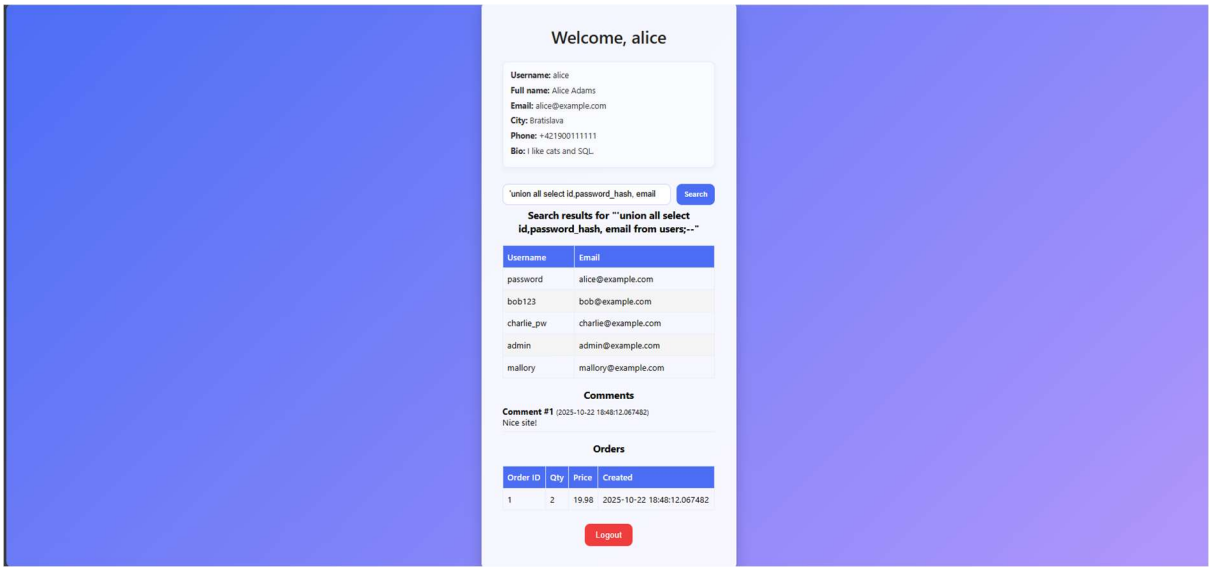



*Figure 3 Search function exploitation*

### 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:

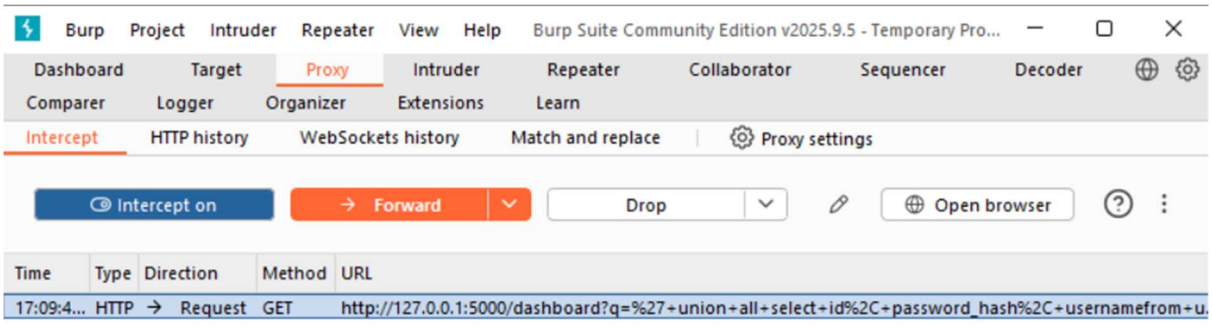
< sqlmap -r request.txt -p <param> --batch --output-dir=results >

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.



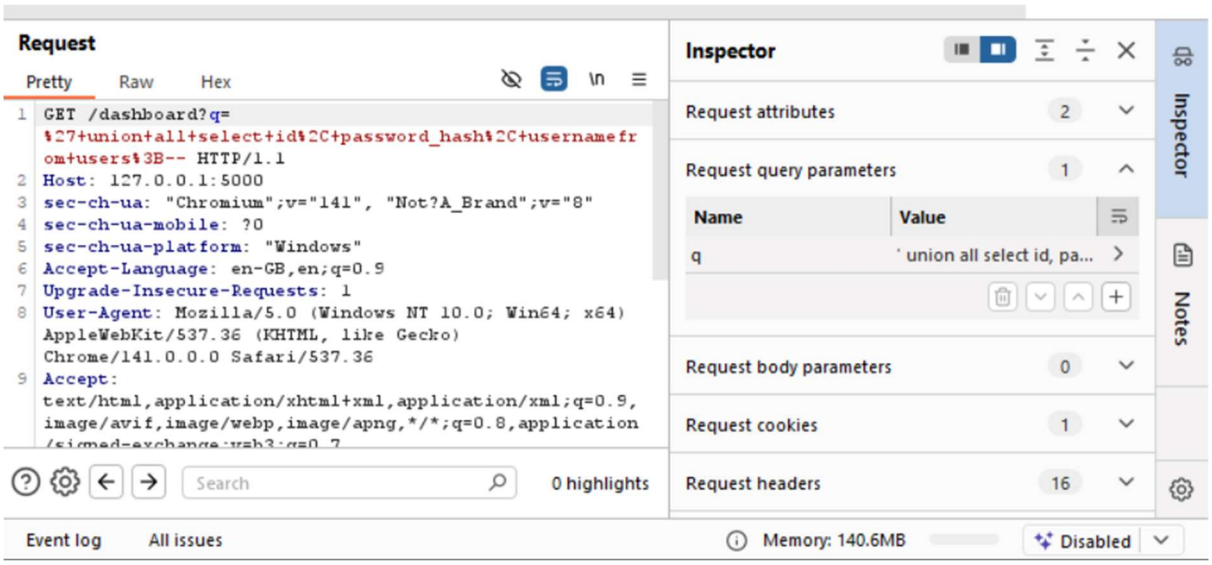



*Figure 4 Burp suite interface and intercept feature*

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.

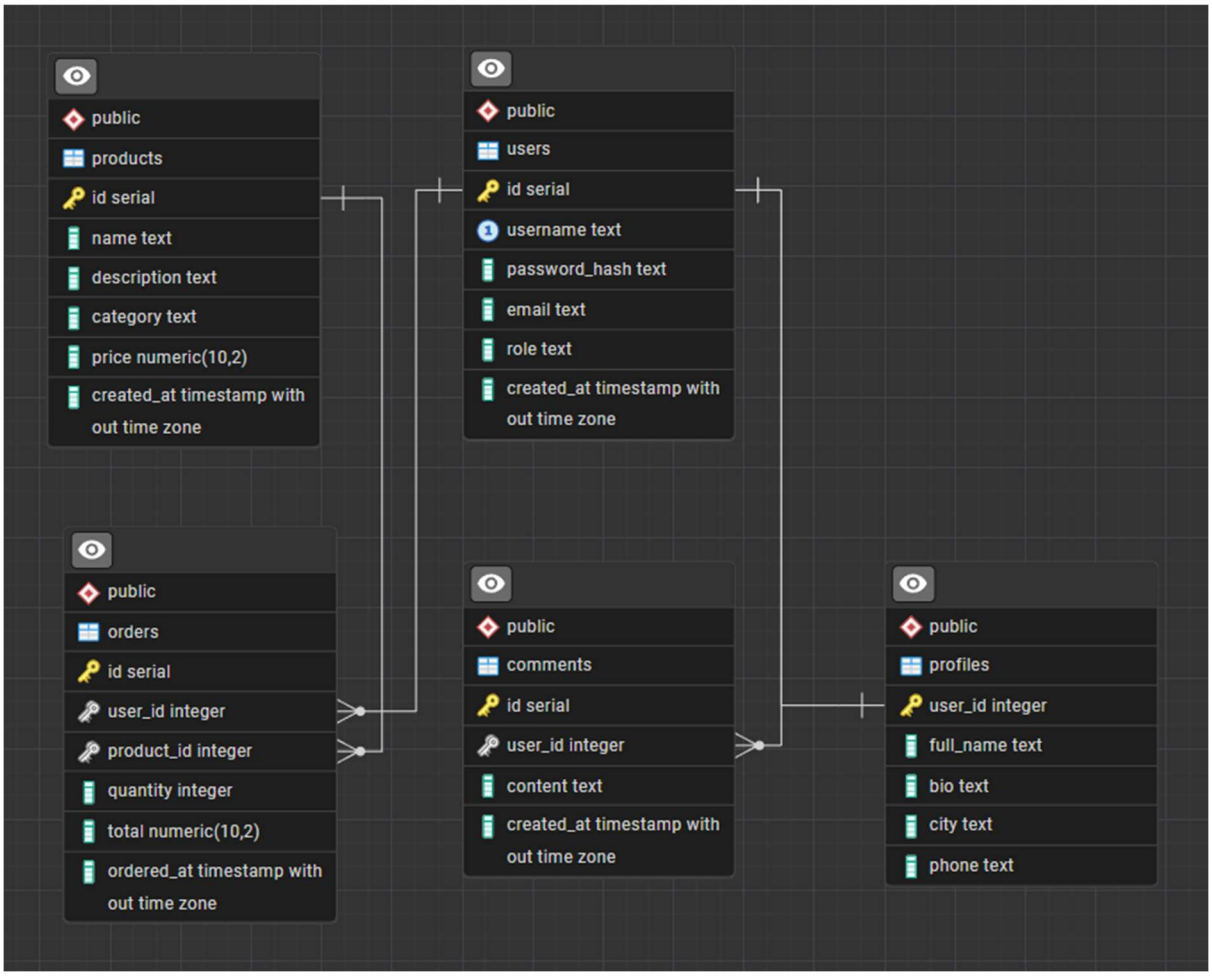



*Figure 5 Schema of the database*

## 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.

I'll use 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. Used resources


- Hussein, Dina & Ibrahim, Dina & Alsalamah, Mona. (2021). A Review Study on SQL Injection Attacks, Prevention, and Detection. 13. 1-9.
- What is SQL Injection (SQLi) and How to Prevent Attacks
- Types of SQL Injection?
- Intercepting HTTP traffic with Burp Proxy - PortSwigger
- sqlmap: automatic SQL injection and database takeover tool
- CWE - CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') (4.18)