SQL Injection Attack and Prevention Project

Prepared by: Ramavath Pavan

B.Tech Pre-Final Year Project

Date: November 16, 2025

Contents

1 Introduction

SQL Injection (SQLi) is one of the most dangerous and well-known web vulnerabilities. It allows attackers to manipulate backend databases by injecting malicious SQL queries through user inputs.

This project demonstrates:

- A **vulnerable PHP + MySQL website**
- A **secure version using prepared statements**
- Three different SQL injection attack surfaces:
  - Login form SQL Injection
  - Search box SQL Injection
  - URL parameter SQL Injection

The goal is to help students understand how SQL Injection works and how to secure applications effectively.

2 Project Setup

2.1 Technologies Used

- PHP 8.x through homebrew(mac)
- MySQL
- HTML + CSS

2.2 Database Structure

Two tables are used:

- **users** — for login
- **products** — for search and product detail pages

2.2.1 Users Table

```
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50),
    password VARCHAR(50)
);

INSERT INTO users (username, password)
VALUES ('admin', 'admin123'), ('user', 'password');
```

2.2.2 Products Table

```
CREATE TABLE products (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100),
    description TEXT,
```

```
    price INT
);

INSERT INTO products (name, description, price) VALUES
('Laptop', 'Gaming laptop', 80000),
('Mobile', '5G smartphone', 20000),
('Headphones', 'Wireless headphones', 3000),
('Keyboard', 'Mechanical keyboard', 2500);
```

3 Module 1: Login SQL Injection

3.1 Vulnerable Login Code

```
$username = $_POST['username'];
$password = $_POST['password'];

$sql = "SELECT * FROM users
        WHERE username='$username' AND password='$password'";

$result = mysqli_query($conn, $sql);
```

3.2 How Attack Works

An attacker enters:

```
username: ' OR '1'='1 --
password: anything
```

This converts the SQL query into:

```
SELECT * FROM users WHERE username='' OR '1'='1 -- ' AND password='anything';
```

Result: LOGIN WITHOUT PASSWORD.

3.3 Secure Login Using Prepared Statements

```
$stmt = $conn->prepare(
    "SELECT * FROM users WHERE username = ? AND password = ?"
);
$stmt->bind_param("ss", $username, $password);
$stmt->execute();
```

Prepared statements prevent attackers from injecting SQL.

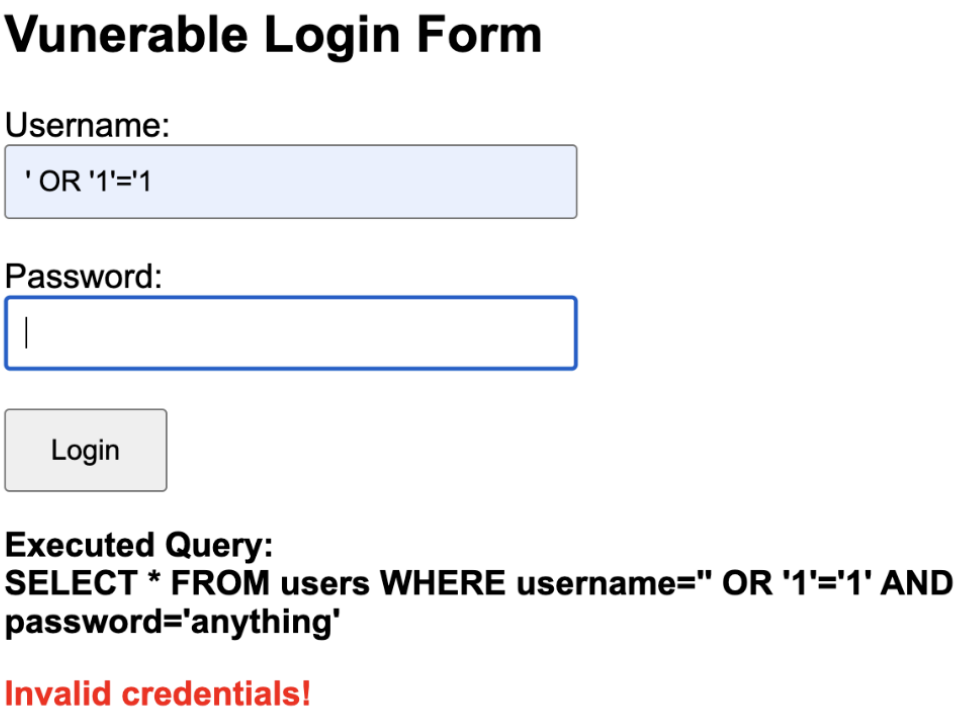

Figure 3.1: Vulnerable Login Bypassed Using payload ' OR '1'='1

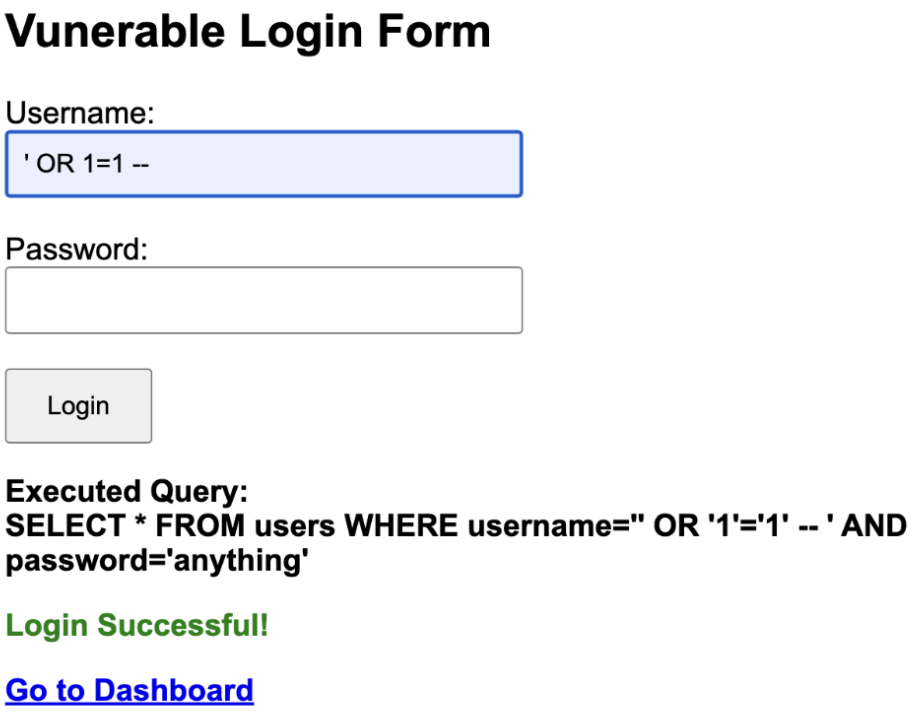

Figure 3.2: Vulnerable Login Bypassed Using payload ' OR 1=1 –

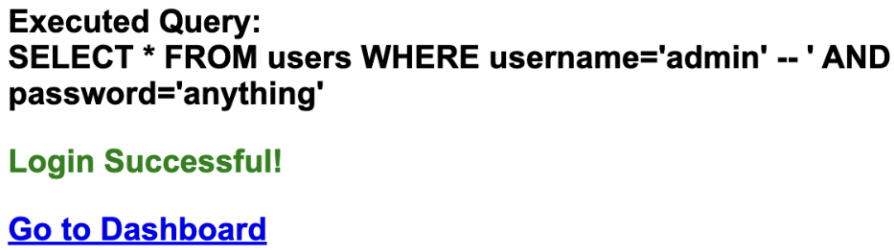

Figure 3.3: Vulnerable Login Bypassed Using payload admin –

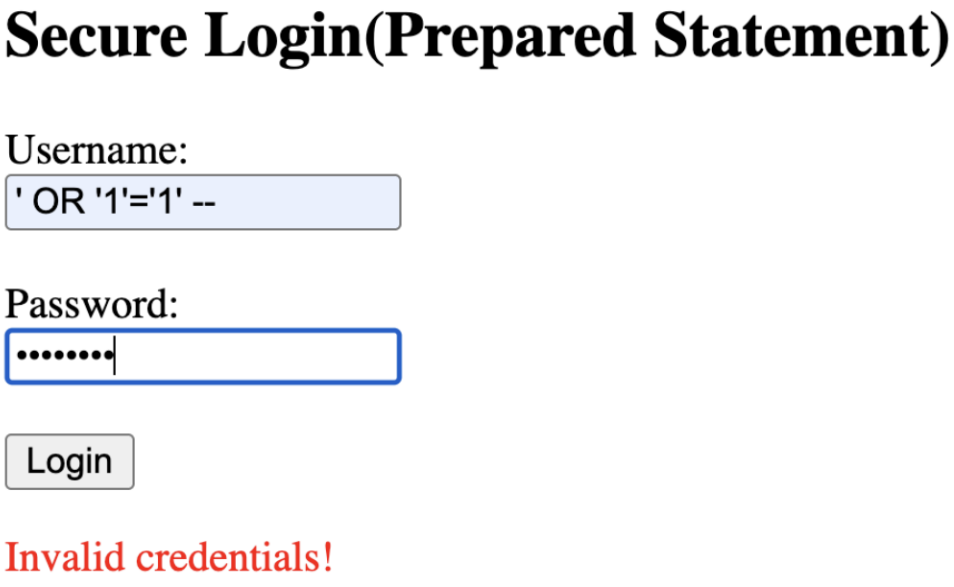

Figure 3.4: Secure login protects sql injection

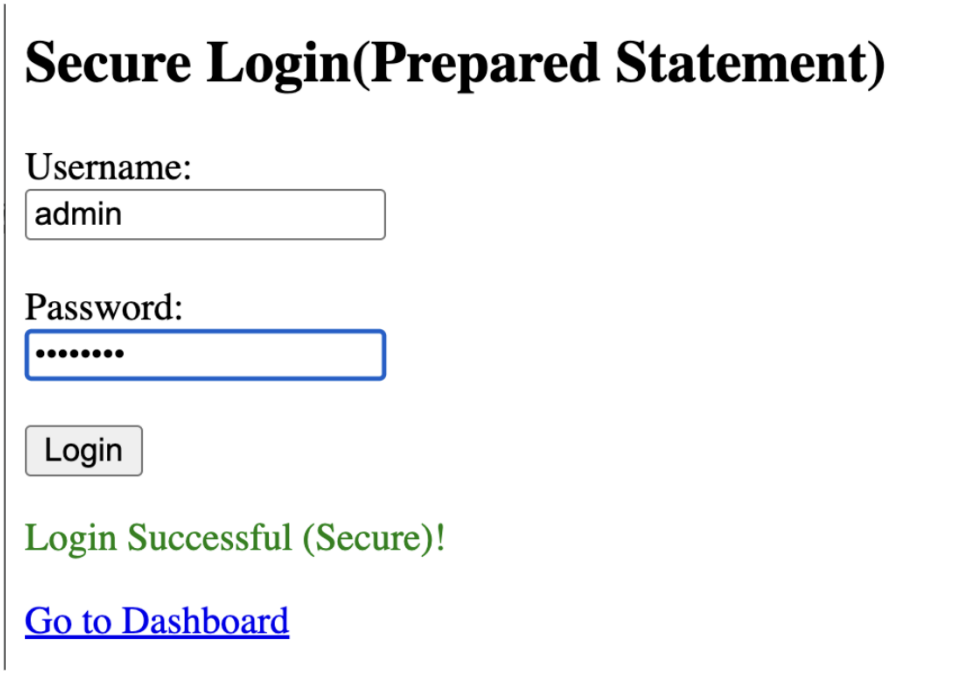

Figure 3.5: Allow authenticated users only

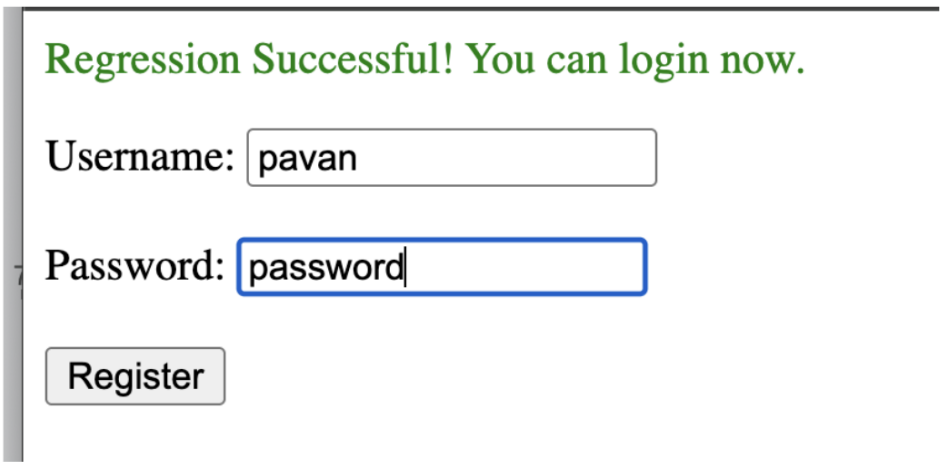

Figure 3.6: registration page Password text converted hash Value

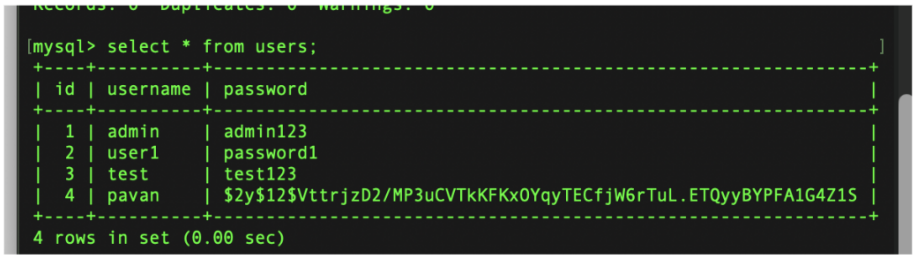

Figure 3.7: Database view

4 Module 2: Search Box SQL Injection (New Feature)

4.1 Vulnerable Search Code

```
$q = $_GET['q'];

$sql = "SELECT * FROM products WHERE name LIKE '%$q%'";

$result = mysqli_query($conn, $sql);
```

4.2 Example Attack

Input:

```
' OR '1'='1 --
```

Effect: Returns ALL products.

4.3 Secure Search Code

```
$q = "%".$_GET['q']."%";

$stmt = $conn->prepare(
    "SELECT * FROM products WHERE name LIKE ?"
);
$stmt->bind_param("s", $q);
$stmt->execute();
```

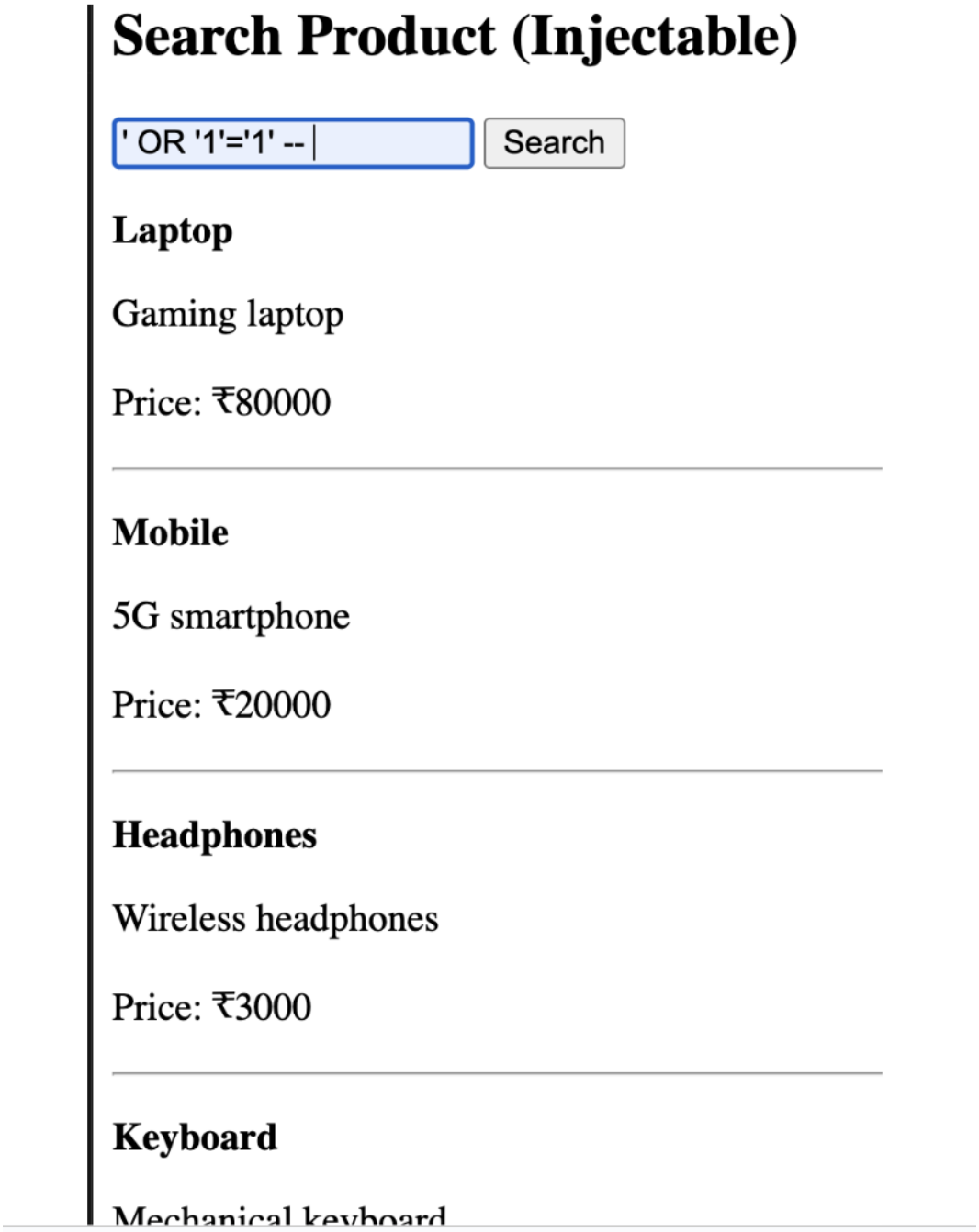

Figure 4.1: Vulnerable search Bypassed Using payload ' OR '1'='1' –

5 Module 3: URL Parameter SQL Injection (New Feature)

5.1 Vulnerable Product Page

```
$id = $_GET['id'];

$sql = "SELECT * FROM products WHERE id = $id";

$result = mysqli_query($conn, $sql);
```

5.2 Example Attack

```
product.php?id=1 OR 1=1
```

This bypasses intended restrictions and dumps all product rows.

5.3 Secure Version

```
$id = $_GET['id'];

$stmt = $conn->prepare(
    "SELECT * FROM products WHERE id = ?"
);
$stmt->bind_param("i", $id);
$stmt->execute();
```

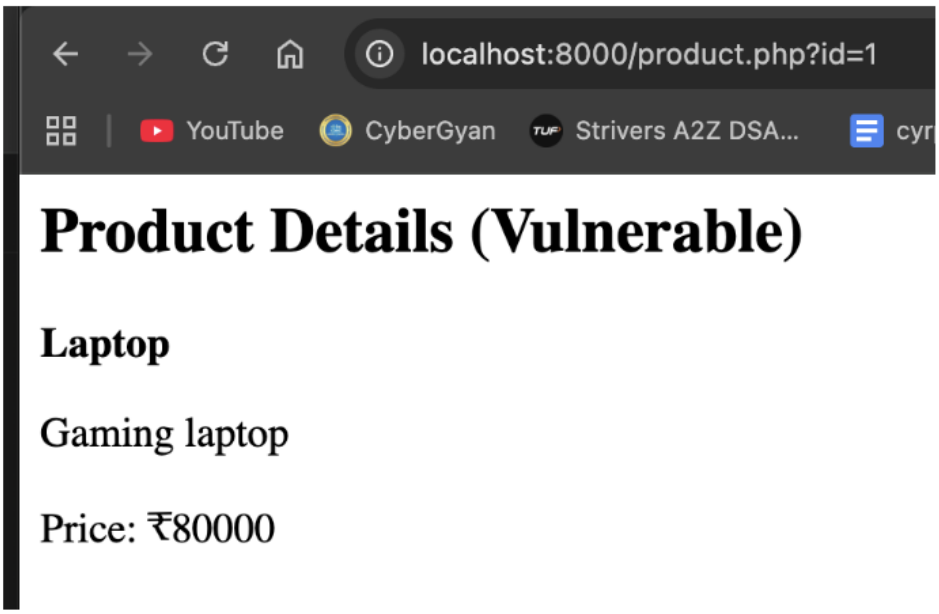

Figure 5.1: product page overview

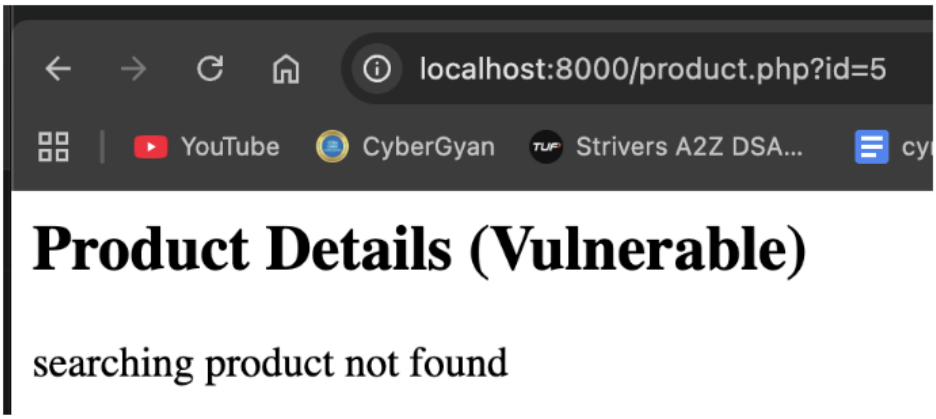

Figure 5.2: URL parameters inserting not known

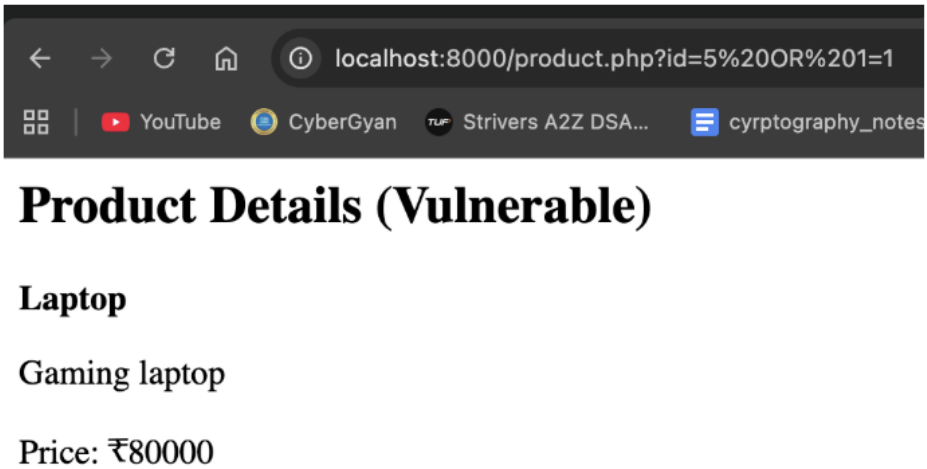

Figure 5.3: URL parameters inseting with payload OR 1=1 at end

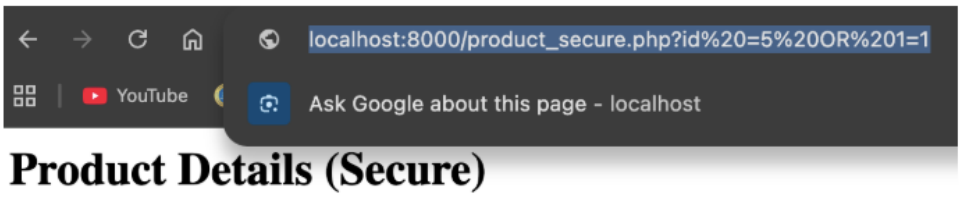

Figure 5.4: Secure url rejects sql injections

6 Security Best Practices

- Use prepared statements (PDO or MySQLi).
- Validate and sanitize all user input.
- Never trust GET/POST data.
- Use least-privilege database accounts.
- Disable detailed SQL error messages.

7 Real-World SQL Injection Examples

SQL Injection is not just an academic topic — it has caused some of the biggest security breaches in history. Below are two famous real-world cases that demonstrate how dangerous SQLi can be.

7.1 Sony Pictures Hack (2011)

In 2011, hackers attacked Sony Pictures using a very simple SQL Injection payload through a web page parameter.

Vulnerable Code Example

```
$id = $_GET['id'];
$query = "SELECT␣*␣FROM␣users␣WHERE␣id␣=␣$id";
```

How Attackers Exploited It

Attackers visited URLs like:

`https://sony.com/user.php?id=1 OR 1=1`

This created an injected SQL query:

```
SELECT * FROM users WHERE id = 1 OR 1=1;
```

Data Hackers Stole

- Usernames
- Passwords
- Emails
- Admin accounts
- Movie details and internal data

Sony lost millions due to this breach.

7.2 TikTok SQL Injection Vulnerability (2020)

In 2020, security researchers discovered a SQL injection flaw in TikTok's user information API.

Endpoint Example

`/user?userId=12345`

Backend Vulnerable Query

```
SELECT * FROM users WHERE id = '$id';
```

Injection Example

`12345 OR 1=1`

This allowed extraction of:

- Private user data
- Phone numbers
- Hidden profiles

TikTok patched the vulnerability quickly, but it highlighted the scale of risk that SQL injection still poses even to major platforms.

8 Conclusion

This project demonstrates:

- Real SQL injection vulnerabilities
- How attackers exploit input fields
- How prepared statements eliminate SQL injection
- Multiple attack surfaces: Login, Search, URL

By creating both vulnerable and secure versions, students gain practical and deep understanding of SQL injection risks and defenses.