A MINI PROJECT REPORT ON

"IMPLEMENT SQL INJECTION VULNERABILITY ATTACK THAT CAUSES THE APPLICATION TO DISPLAY DETAILS OF ALL THE PRODUCTS AVAILABLE ON WEBSITE."

Submitted By

Student Name	Roll No
Omkar Biradar	A-20
Sujal Erande	A-45
Shivam Gavali	A-48
Yashwant Ghorband	A-53

Under The Guidance Of

Prof. Pooja Dehankar

IN THE FULFILLMENT OF

CYBER SECURITY MINI PROJECT REPORT

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE ENGINEERING

A. Y. 2024-25

Ajeenkya D. Y. Patil School of Engineering,

Lohegaon, Pune – 412 105

DEPARTMENT OF ARTIFICIAL INTELLIGENCE & DATA SCIENCE ENGINEERING

CERTIFICATE

This is to certify that,

Omkar Biradar A-20
Sujal Erande A-45
Shivam Gavali A-48
Yashwant Ghorband A-53

from Ajeenkya D. Y. Patil School of Engineering Institute has completed mini project of Third year engineering entitled "**Implement SQL Injection Vulnerability Attack That Causes The Application To Display Details Of All The Products Available On Website.**" during the academic year 2024-2025.The project completed in group consisting of 03 persons under the guidance of the Faculty Guide.

Date: 25/04/2025

Prof. Pooja Dehankar	**Dr. Bhagyashree Dhakulkar**	**Dr. F. B. Sayyad**
Guide	**HOD-AI&DS**	**Principal**

ACKNOWLEGEMENT

It gives me great pleasure and immense satisfaction to present this special Seminar report on **"IMPLEMENT SQL INJECTION VULNERABILITY ATTACK THAT CAUSES THE APPLICATION TO DISPLAY DETAILS OF ALL THEPRODUCTS AVAILABLE ON WEBSITE."**, which is the result of the unwavering support, expert guidance, and focused direction of my guide **Prof. Pooja Dehankar** to whom I express my deep sense of gratitude and humble thanks, for her valuable guidance throughout the presentation work. The success of this Seminar has throughout depended upon an exact blend of hard work and unending cooperation and guidance, extended to me by the superiors at our college.

Furthermore, I am indebted to **Dr. Bhagyashree Dhakulkar,** HoD of Artificial Intelligence and Data Science Engineering and **Dr. Farook Sayyad,** the Principal whose constant encouragement and motivation inspired me to do my best.

Last but not least I sincerely thank to my colleagues, the staff, and all others who directly or indirectly helped us and made numerous suggestions that have surely improved the quality of my work.

TABLE OF CONTENT

ABSTRACT

In today's digital landscape, web application security plays a crucial role in protecting sensitive user and organizational data. One of the most common and dangerous vulnerabilities faced by web applications is **SQL Injection (SQLi)**. This project focuses on practically demonstrating the SQL Injection vulnerability, its consequences, and the methods to prevent it. In the project, a **deliberately insecure web application** is developed using **Flask** (a Python micro-framework) and a **SQLite** database. The application allows users to search for and modify product details by submitting a product ID through a web form. The initial version of the application constructs SQL queries dynamically by directly embedding user input into the SQL statement without any sanitization. This practice leaves the application wide open to SQL Injection attacks. Attackers can exploit the vulnerability by entering malicious inputs such as logical operators (`OR 1=1`) to retrieve unauthorized data or modify multiple records at once. For example, an attacker could update the names of all products in the database or even delete the entire product list by crafting specific malicious inputs. These demonstrations clearly show the **severity of SQL Injection attacks**, even on small and simple applications. The project further emphasizes the **importance of secure coding practices** by implementing a secure version of the same application using **parameterized queries** (also known as prepared statements). Parameterized queries separate the SQL code from the data, thereby preventing attackers from altering the structure command. By comparing the vulnerable and secured versions of the application side-by-side, the project highlights how a simple change in query construction can protect against potentially devastating attacks. Additionally, the project discusses the limitations of SQLite in executing multiple statements at once, which partially restricts certain types of SQLi but does not eliminate the threat of logical manipulation (`OR 1=1`) attacks. The project also proposes best practices for further hardening web applications, such as input validation, least privilege principle, and proper error handling. Ultimately, the project serves as an educational tool to build awareness about web security, with an emphasis on **developing secure applications from the start**, rather than patching vulnerabilities after they are exploited.

Keywords: SQL Injection (SQLi), Cybersecurity, Web Application Security, Vulnerabilities, Flask Framework, SQLite Database, Parameterized Queries, Input Validation, Secure Coding Practices, Database Security, Attack Demonstration, User Input Sanitization.

INTRODUCTION

Web applications are increasingly used for a wide range of critical activities, including e-commerce, online banking, healthcare management, and education portals. As the reliance on web technologies grows, so does the importance of ensuring their security. [1] One of the most dangerous and commonly found vulnerabilities in web applications is **SQL Injection (SQLi)**.

[4] **SQL Injection** is a code injection technique that exploits a security vulnerability in an application's database layer. It occurs when user input is improperly sanitized and directly embedded into SQL queries, allowing attackers to manipulate the structure and execution of the query. As a result, malicious users can retrieve sensitive data, modify database contents, or even take full control over the database system without proper authorization.

[5] The primary goal of this project is to **demonstrate how SQL Injection vulnerabilities arise** in a real-world web application, how an attacker can exploit them, and how developers can **securely code their applications to prevent such attacks**.

In this project, a **simple web application** is created using the **Flask** framework and a **SQLite** database. The application allows users to search for products by entering a Product ID into a form. [2] In the insecure version, the user's input is directly inserted into an SQL query without any filtering or validation. This opens the door for various forms of SQL Injection attacks, such as:

- Displaying all product records with inputs like `1 OR 1=1`.
- Modifying or deleting records by injecting additional SQL commands.

The vulnerable code is intentionally designed to **simulate a real-world security flaw** so that learners can observe and understand the actual risks associated with SQL Injection.

[6] After demonstrating the attack, the project then **rebuilds the application in a secure manner**. The secure version uses **parameterized queries**, which prevent malicious user input from altering the SQL command's structure. This approach ensures that inputs are treated as data only and not executable SQL code.

[3] The project not only covers the technical aspects of SQL Injection attacks but also emphasizes **best practices** for secure web development:

- Always validate and sanitize user inputs.
- Use prepared statements (parameterized queries) instead of dynamic SQL generation.
- Handle database errors securely without exposing sensitive system information.
- Apply the principle of least privilege for database access.

Through this project, developers and students can gain a **practical understanding of vulnerabilities**, **how to exploit them safely for learning purposes**, and **how to effectively defend against them**. It provides a strong foundation in secure coding practices and encourages proactive thinking toward cybersecurity during software development.

In today’s rapidly evolving digital landscape, [4] the knowledge of vulnerabilities like SQL Injection and their prevention is critical for creating **reliable, robust, and secure web applications**.

OBJECTIVES

The main objectives of this project, titled **"Demonstration of SQL Injection Attack and Its Prevention,"** are:

1. **Understand the Concept of SQL Injection:**
    - To study what SQL Injection is, how it occurs, and why it poses a major security threat to web applications.
2. **Develop a Vulnerable Web Application:**
    - To create a simple web application using **Flask** and **SQLite** where user input is not sanitized, intentionally leaving it vulnerable to SQL Injection attacks.
3. **Demonstrate SQL Injection Attacks Practically:**
    - To perform SQL Injection attacks on the vulnerable application by injecting malicious SQL commands.
    - To show how unauthorized users can manipulate queries to **bypass authentication**, **retrieve confidential data**, **modify records**, or **corrupt the database**.
4. **Analyze the Impact of SQL Injection:**
    - To evaluate the real-world consequences of successful SQL Injection attacks, including data breaches, loss of integrity, and system compromise.
5. **Implement Security Measures Against SQL Injection:**
    - To rebuild the application securely by using **parameterized queries (prepared statements)** and **input validation** techniques.
    - To demonstrate how minor code changes can effectively mitigate serious vulnerabilities.
6. **Promote Secure Coding Practices:**
    - To encourage developers to follow **best practices** in web application development such as input sanitization, principle of least privilege, and proper error handling.
7. **Raise Awareness About Web Application Security:**
    - To emphasize the importance of considering security from the early stages of software development, rather than treating it as an afterthought.
8. **Hands-on Learning Approach:**
    - To provide a practical, experiential learning platform where students and developers can directly observe vulnerabilities and apply defenses themselves.

SCOPE

This project focuses on exploring and addressing the critical security vulnerability known as **SQL Injection (SQLi)** within web applications. The scope of the project is outlined as follows:

1. **Development of a Vulnerable Web Application:**
    - Building a simple yet functional web application using **Flask** (Python framework) and **SQLite** database.
    - Implementing basic functionalities like **product search** based on **user input** (Product ID) without proper input sanitization, intentionally leaving the system vulnerable to SQL Injection attacks.
2. **Demonstration of SQL Injection Attacks:**
    - Performing **realistic SQL Injection attacks** on the vulnerable application.
    - Demonstrating how attackers can:
        - Retrieve sensitive data (e.g., details of all products).
        - Bypass normal application behavior.
        - Modify or corrupt database content maliciously.
3. **Security Analysis and Risk Evaluation:**
    - Analyzing the extent of damage SQL Injection can cause if exploited by an attacker.
    - Highlighting how small vulnerabilities can lead to **data breaches**, **reputational damage**, and **financial losses**.
4. **Implementation of Secure Coding Practices:**
    - Redesigning the application to **prevent SQL Injection** by:
        - Using **parameterized queries** instead of direct SQL string concatenation.
        - Validating and sanitizing user input properly.
        - Handling errors securely without exposing system details.
5. **Educational and Awareness Purpose:**
    - Educating developers, students, and security enthusiasts about the dangers of SQL Injection.
    - Demonstrating both the **offensive (attacking)** and **defensive (securing)** sides of web application development.
6. **Limitations Defined:**
    - The project is confined to a basic web application model and **does not cover** more complex attack scenarios like:
        - Blind SQL Injection.
        - Out-of-band SQL Injection.
        - Advanced exploitation methods against large enterprise-grade applications.
    - The database system used is **SQLite**, and advanced SQLi impacts on other databases like **MySQL**, **PostgreSQL**, or **MS SQL Server** are outside the current project scope.
7. **Future Scope:**
    - Extending the project to cover **more complex attacks** (Blind SQLi, Time-based attacks).
    - Studying SQL Injection impacts on **different DBMS platforms**.
    - Implementing **Web Application Firewalls (WAFs)** and other network-level defenses.
    - Applying **ORMs (Object Relational Mappers)** like SQLAlchemy for safer database interaction.

PROBLEM STATEMENT

“This project addresses the vulnerability of web applications to SQL Injection attacks due to improper user input handling. It demonstrates how attackers can exploit this flaw to access or modify sensitive database information. The project further focuses on implementing secure coding practices, such as parameterized queries, to prevent such attacks and ensure database security.”

PROJECT DETAILS

METHODOLOGY:

1. Develop a Vulnerable Web Application:

- Create a simple Flask web application connected to an SQLite database.
- Implement a product search feature using **unsanitized user input** directly in SQL queries to intentionally leave it vulnerable.

2. Perform SQL Injection Attack:

- Inject malicious SQL inputs through the search field to display or manipulate product data, demonstrating the real-world impact of SQL Injection vulnerabilities.

3. Analyze the Vulnerability:

- Observe how the application behavior changes due to SQL Injection.
- Identify how attackers can retrieve, alter, or damage data.

4. Implement Security Measures:

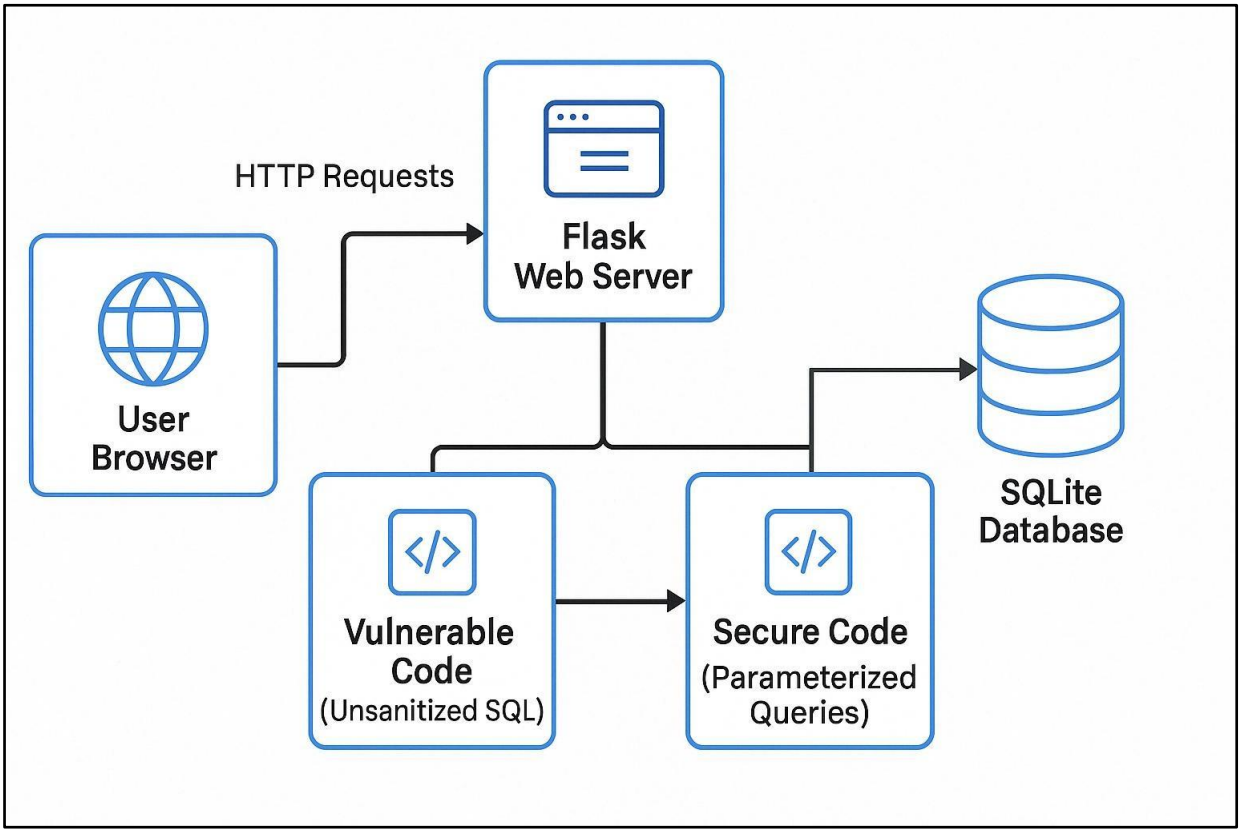
- Modify the application code to **use parameterized queries** and **input validation** to secure the database interactions.
- Retest to ensure the application is **protected against SQL Injection**.

5. Document the Findings:

- Prepare a detailed report highlighting the attack process, the vulnerabilities found, and the secure coding practices applied to prevent future attacks.

DESIGN / IMPLEMENTATION:

Architecture:



1. Vulnerable Web Application Design

The web application was built using **Flask** as the web framework and **SQLite** as the database. The application has a basic **product search** functionality where the user can input a **product ID**, and the [6] application queries the database to fetch product details.

However, in this vulnerable version, the application directly inserts the user input into the SQL query without proper sanitization, allowing attackers to perform **SQL Injection**.

Code: Vulnerable Version

```
from flask import Flask, request, render_template_string
import sqlite3

app = Flask(__name__)

HTML_TEMPLATE = '''
<h2>Search Product</h2>
<form method="POST">
    Product ID: <input type="text" name="product_id">
    <input type="submit" value="Search">
```

```
</form>

{% if product %}
    <h3>Product Details:</h3>
    {% for p in product %}
        <p><b>Name:</b> {{ p[0] }}</p>
        <p><b>Description:</b> {{ p[1] }}</p>
        <p><b>Price:</b> ${{ p[2] }}</p>
        <hr>
    {% endfor %}
{% elif error %}
    <p style="color: red;">{{ error }}</p>
{% endif %}
'''

@app.route('/', methods=['GET', 'POST'])
def index():
    product = None
    error = None

    if request.method == 'POST':
        product_id = request.form['product_id']

        # Vulnerable Code: Unsanitized SQL query
        query = f"SELECT name, description, price FROM products WHERE id =
{product_id}"

        try:
            conn = sqlite3.connect('products.db')
            cursor = conn.cursor()
            cursor.execute(query)
            product = cursor.fetchall()
            conn.close()

            if not product:
                error = "No product found with that ID."


        except Exception as e:
            error = f"Error: {str(e)}"

    return render_template_string(HTML_TEMPLATE, product=product,
error=error)

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

2. Demonstrating SQL Injection Attack

The vulnerable application allows an attacker to inject malicious SQL code. For instance, by inputting the following into the product ID field:

```
sql
CopyEdit
1 OR 1=1
```

This would modify the SQL query to:

```
sql
CopyEdit
SELECT name, description, price FROM products WHERE id = 1 OR 1=1
```

This query would return **all products** in the database, bypassing the intended behavior of searching for a specific product by ID.

Attack Output Example:

If the attacker inputs `1 OR 1=1`, the application would return all product data, even if they are not authorized to see it.

Example output from the application:

```
yaml
CopyEdit
Product Details:
Name: Product 1
Description:   A   great   product.
Price: $25.00

Name: Product 2
Description:   Another   amazing   product.
Price: $40.00
...
```

3. Secure Version (Prevention of SQL Injection)

To prevent SQL Injection, we use **parameterized queries** with **placeholders** for user input. This ensures that user inputs are treated as data, not executable code.

Code: Secure Version

```
python
CopyEdit
from flask import Flask, request, render_template_string
import sqlite3

app = Flask(__name__)


HTML_TEMPLATE = '''
<h2>Search Product</h2>
<form method="POST">
    Product ID: <input type="text" name="product_id">
    <input type="submit" value="Search">
</form>

{% if product %}
    <h3>Product Details:</h3>
    {% for p in product %}
        <p><b>Name:</b> {{ p[0] }}</p>
        <p><b>Description:</b> {{ p[1] }}</p>
        <p><b>Price:</b> ${{ p[2] }}</p>
        <hr>
{% endfor %}
{% elif error %}
    <p style="color: red;">{{ error }}</p>
 {% endif %}
 '''

 @app.route('/', methods=['GET', 'POST'])
 def index():
```

```
    product = None
    error = None

    if request.method == 'POST':
        product_id = request.form['product_id']

        # Secure Code: Parameterized SQL query
        query = "SELECT name, description, price FROM products WHERE id = ?"

        try:
            conn = sqlite3.connect('products.db')
            cursor = conn.cursor()
            cursor.execute(query, (product_id,))
            product = cursor.fetchall()
            conn.close()

            if not product:
                error = "No product found with that ID."

        except Exception as e:
            error = f"Error: {str(e)}"

    return render_template_string(HTML_TEMPLATE, product=product,
error=error)

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

4. Secure Version Output:

With the secure version, an attacker cannot perform SQL Injection. The input `1 OR 1=1` will only be treated as a string and will not alter the query's behavior.

If the user enters `1 OR 1=1`:

```
sql
CopyEdit
SELECT name, description, price FROM products WHERE id = '1 OR 1=1'
```

This query will safely return the product with **ID = 1** and **not all products**.

Output Example (Secure Version):

```
yaml
CopyEdit
Product Details:
Name: Product 1
Description: A great product.
Price: $25.00
```

5. Database Configuration:

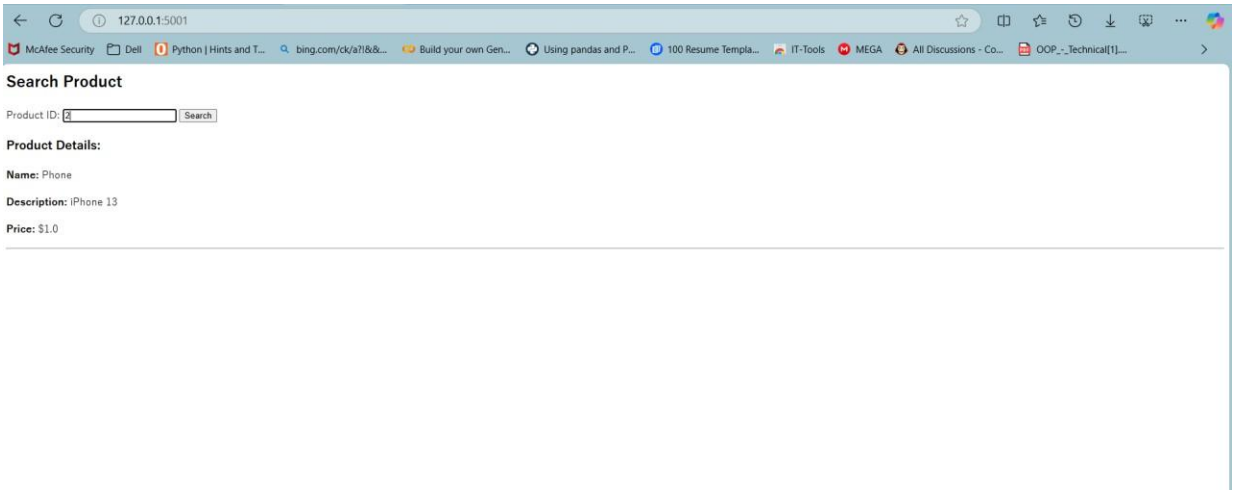
The **SQLite database** used in the project has the following schema for the `products` table:

```
sql
CopyEdit
CREATE TABLE products (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    description TEXT,
    price REAL
);
```

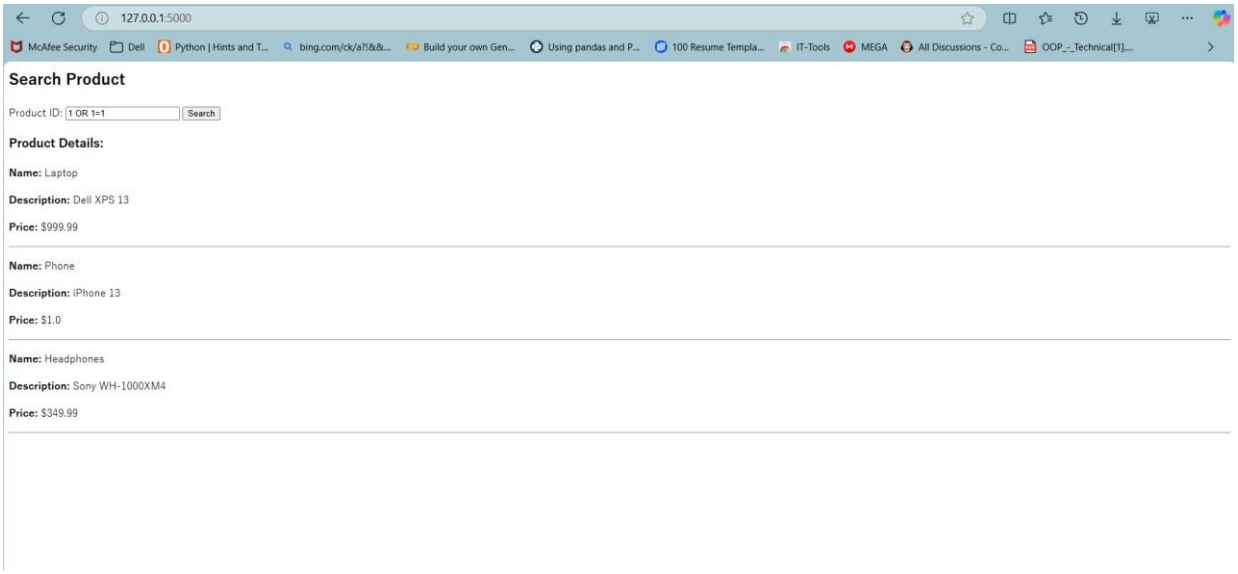
The database contains product records, and the application fetches product details based on the provided ID.

GRAPHICAL USER INTERFACE

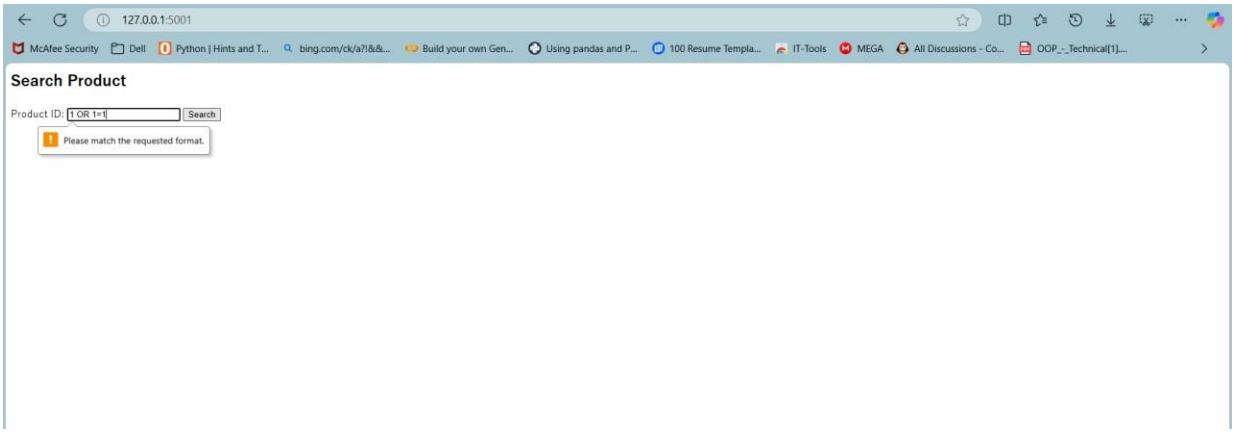
1. Valid Query (Without SQL Injection Attack)



2. Make Attack using SQL Injection



3. Secure parameterized query



Code

- Database Setup Code

```python
import sqlite3

# Connect to database
conn = sqlite3.connect('products.db')
c = conn.cursor()

# Create products table
c.execute('''
CREATE TABLE IF NOT EXISTS products (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    description TEXT,
    price REAL
)
''')

# Insert sample data
products = [
    ('Laptop', 'Dell XPS 13', 999.99),
    ('Phone', 'iPhone 13', 799.99),
    ('Headphones', 'Sony WH-1000XM4', 349.99)
]

c.executemany('INSERT INTO products (name, description, price) VALUES (?, ?, ?)', products)

conn.commit()
conn.close()

print("Database setup complete!")

```

- VULNERABLE CODE: unsanitized SQL query

```python
from flask import Flask, request, render_template_string
import sqlite3

app = Flask(__name__)

HTML_TEMPLATE = '''
<h2>Search Product</h2>
<form method="POST">
    Product ID: <input type="text" name="product_id">
    <input type="submit" value="Search">
</form>

{% if product %}
    <h3>Product Details:</h3>
    {% for p in product %}
        <p><b>Name:</b> {{ p[0] }}</p>
        <p><b>Description:</b> {{ p[1] }}</p>
        <p><b>Price:</b> ${{ p[2] }}</p>
        <hr>
    {% endfor %}
{% elif error %}
    <p style="color: red;">{{ error }}</p>
{% endif %}

'''

@app.route('/', methods=['GET', 'POST'])
def index():
    product = None
    error = None

    if request.method == 'POST':
        product_id = request.form['product_id']

        # ❗ VULNERABLE CODE: unsanitized SQL query
        query = f"SELECT name, description, price FROM products WHERE id = {product_id}"

        try:
            conn = sqlite3.connect('products.db')
            cursor = conn.cursor()
            cursor.execute(query)
            product = cursor.fetchall()
            conn.close()

            if not product:
                error = "No product found with that ID."

        except Exception as e:
            error = f"Error: {str(e)}"

    return render_template_string(HTML_TEMPLATE, product=product, error=error)

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

- SECURE CODE: sanitized SQL query

```python
from flask import Flask, request, render_template_string
import sqlite3

app = Flask(__name__)

HTML_TEMPLATE = '''
<h2>Search Product</h2>
<form method="POST">
    Product ID: <input type="text" name="product_id" required pattern="\\d+">
    <input type="submit" value="Search">
</form>

{% if product %}
    <h3>Product Details:</h3>
    {% for p in product %}
        <p><b>Name:</b> {{ p['name'] }}</p>
        <p><b>Description:</b> {{ p['description'] }}</p>
        <p><b>Price:</b> ${{ p['price'] }}</p>
        <hr>
    {% endfor %}
{% elif error %}
    <p style="color: red;">{{ error }}</p>
{% endif %}
'''

def get_db_connection():
    conn = sqlite3.connect('products.db')
    conn.row_factory = sqlite3.Row  # So you can access columns by name
    return conn

@app.route('/', methods=['GET', 'POST'])
def index():
    product = None
    error = None

    if request.method == 'POST':
        product_id = request.form['product_id']

        # Extra input validation (only digits)
        if not product_id.isdigit():
            error = "Invalid product ID."
            return render_template_string(HTML_TEMPLATE, product=product, error=error)

        try:
            conn = get_db_connection()
            cursor = conn.cursor()

            # ✅ Secure parameterized query
            cursor.execute('SELECT name, description, price FROM products WHERE id = ?', (product_id,))
            product = cursor.fetchall()

            conn.close()

            if not product:
                error = "No product found with that ID."

        except Exception as e:
            error = "Internal server error. Please try again later."
            # (Optional: log the actual error for debugging)

    return render_template_string(HTML_TEMPLATE, product=product, error=error)

if __name__ == "__main__":
    app.run(debug=True, port=5001)
```

CONCLUSION

This project effectively demonstrates the critical impact of SQL Injection vulnerabilities in web applications. By intentionally designing a vulnerable system, we showcased how attackers can exploit unsanitized user inputs to access, modify, or destroy sensitive database information without proper authorization. Through hands-on practical examples, we observed how even the simplest form of input manipulation, such as inserting logical operators or malicious SQL code, can completely bypass application logic, retrieve confidential information, and compromise the integrity of the entire database. This not only leads to breaches of data confidentiality but also threatens data availability and authenticity, causing potential reputational damage and financial loss to organizations. Moreover, the project strongly highlights the significance of adopting secure coding practices from the earliest stages of software development. In particular, it emphasizes the use of parameterized queries as a fundamental defense mechanism to safeguard applications against SQL Injection attacks. Parameterized queries prevent malicious user input from interfering with the intended structure of SQL statements, thus eliminating one of the most common security loopholes in web applications. Throughout the project, it became clear that security cannot be treated as an optional or secondary aspect of application development; instead, it must be a core part of the design and implementation process. Proper input validation, error handling, minimal privilege assignment to database users, and a strong focus on code security can drastically reduce the risk of exploitation. Overall, this project serves as a valuable learning experience, demonstrating that even seemingly small security oversights can lead to major vulnerabilities. It reinforces the crucial lesson that building robust, secure, and trustworthy applications requires a proactive approach toward identifying and mitigating risks, fostering a culture of secure development practices, and continuously evolving to counter emerging cybersecurity threats.

REFERENCES

[1] Halfond, W. G. J., Viegas, J., & Orso, A. (2006).
A classification of SQL-injection attacks and countermeasures.
Proceedings of the IEEE International Symposium on Secure Software Engineering.

[2] Su, Z., & Wassermann, G. (2006).
The Essence of Command Injection Attacks in Web Applications.
Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).

[3] Bau, J., Bursztein, E., Gupta, D., & Mitchell, J. (2010).
State of the Art: Automated Black-Box Web Application Vulnerability Testing.
Proceedings of the IEEE Symposium on Security and Privacy (SP).

[4] Fonseca, J., Vieira, M., & Madeira, H. (2007).
Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks.
Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing (PRDC).

[5] Anley, C. (2002).
Advanced SQL Injection in SQL Server Applications.
NGSSoftware Insight Security Research (NISR) White Paper.

[6] OWASP Foundation. (2017).
OWASP Top 10 - 2017: The Ten Most Critical Web Application Security Risks.
OWASP Foundation.