UNIVERSITÀ
DEGLI STUDI
FIRENZE

PENETRATION TESTING REPORT

CONDUCTED BY FARMASTER©
CLIENT: MICHELANGELO CAR DEALERSHIP

SCHOOL OF MATHEMATICAL, PHYSICAL AND NATURAL SCIENCES
SOFTWARE: SCIENCE AND TECHNOLOGY
B032431 (B255) - PENETRATION TESTING 2024-2025

Syed Mohammad Askari Abidi (Farmaster©)
Ethical Hacker – Matriculation: 7178458
Email: syed.abidi@edu.unifi.it

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Table of Contents

UNIVERSITÀ DEGLI STUDI FIRENZE

Executive Summary

This report presents the results of a penetration test conducted on the **Michelangelo Car Dealership** internal web application hosted in a Docker container. The goal of the assessment was to identify exploitable security flaws that could be abused by an unauthenticated attacker simulating a black-box testing scenario.

Summary of Results

A critical SQL Injection vulnerability was discovered in the login form (/login.php), which enabled an authentication bypass using a crafted input. Initial testing triggered visible MariaDB syntax errors, confirming the injection point. A payload using `' OR '1' LIKE '1' LIMIT 1 #` successfully bypassed the login mechanism and granted unauthorized access to the system.

This report outlines the methodology, reproduction steps, payloads used, and a remediation plan to address the identified security flaw. The vulnerability has been classified as Critical (CVSS v3.1 score: 9.8) and requires urgent attention according to the given table.

Risk Level	Description	Potential Impact
Urgent	Involves vulnerabilities like remote code execution, full file system access, and stealth mechanisms (e.g., trojans, backdoors).	Complete system takeover with root/admin privileges; full control of the target host. Immediate action required.
Critical	Exposes the system to file reading vulnerabilities, limited writing capabilities, and potential remote access vectors.	Attackers can access confidential files or deploy limited payloads. High risk of data compromise and partial system manipulation.
High	Includes directory traversal, denial of service (DoS), and partial data exposure.	May lead to unauthorized reading of sensitive data, system crashes, or security bypasses. Could be used as a steppingstone for more severe attacks.
Medium	Discloses non-critical but useful technical data such as server configurations, patch levels, and software versions.	Helps attackers map the environment and plan targeted future exploits. Not immediately dangerous but aids reconnaissance.
Low	Reveals basic system information such as HTTP headers, open ports, or non-sensitive banner details.	Offers minimal security but can be used for fingerprinting and inventory mapping. Generally informational.

Methodologies Used

The penetration testing was conducted using a manual black-box approach. No access to the source code or backend systems was available, and all interactions were performed through the application's web interface at http://localhost:8080/. The assessment focused on identifying vulnerabilities in authentication, input validation, and backend query handling. All tests were executed in a controlled local environment, simulating a real-world external attacker.

The following methodology was applied:

Reconnaissance

The application was manually explored to understand its structure and input vectors. Pages such as /login.php, /detail.php, and /recovery.php were accessed to identify points of user interaction.

Key steps:

1. Navigated through visible pages and links in the application.
2. Identified key input fields and URL parameters.
3. Took note of server error messages and application behavior.

Input Validation Testing

Input fields were tested with crafted characters to analyze how the application processes user-supplied data. These characters were used to test for unfiltered input that could lead to injection flaws.

Characters tested included:

1. ' (single quote)
2. -- (SQL comment)
3. #, " and basic SQL keywords (OR, LIKE, SELECT)

These tests were primarily performed on the login form and URL query strings. The appearance of raw SQL error messages confirmed the lack of proper input sanitization.

SQL Injection Testing

Focused testing on the login form (/login.php) was conducted to confirm and exploit SQL injection vulnerabilities. Multiple variations of payloads were manually injected into the username and password fields.

Examples of tested payloads:

1. ' OR 1=1 --
2. ' OR '1' LIKE '1' LIMIT 1 #
3. ' OR 1=1 #

Eventually, the injection ' OR '1' LIKE '1' LIMIT 1 # was successful in bypassing login and authenticating the tester without valid credentials.

Authentication Bypass Attempt

Using the successful payload, the tester was able to access the system beyond the login page. This confirmed that the backend logic relied on unsanitized user input in SQL queries, and no prepared statements were in place.

This step provided full unauthorized access using a single SQL injection attack vector.

Password Hash Analysis

During earlier failed attempts, the application revealed SQL error messages that included raw MD5 password hashes. These hashes were copied and decoded using public MD5 hash lookup tools.

For example:

- Hash found: 3979f7f001b2962787ccc75f394b7689
- Resolved plaintext: 123

This revealed the use of weak password hashing without salting or proper security measures.

Documentation of Findings

All confirmed vulnerabilities were recorded with:

1. The affected component (e.g., login page).
2. The exact payload used.
3. Screenshots of successful exploitation.
4. Description of the vulnerability.
5. Risk assessment and CVSS scoring.
6. Recommended remediation steps.

Findings

FINDING1: SQL INJECTION – LOGIN BYPASS

Description

A critical SQL injection vulnerability was identified in the login form of the Michelangelo Car Dealership web application. The application fails to sanitize user input before embedding it directly into an SQL query. This allows an attacker to manipulate the query logic and bypass authentication entirely.

The login mechanism accepts crafted SQL payloads in the username field, which results in a manipulated query that returns a valid user record without verifying legitimate credentials. This behavior confirms that the application is vulnerable to classical SQL injection.

Severity

CRITICAL (CVSS V3.1 BASE SCORE: 9.8)

This vulnerability allows unauthenticated attackers to gain access to the application without knowing any valid username or password.

Affected Component

- **Page**: /login.php
- **Vulnerable Field**: username input field.
- **Technology Involved**: PHP + MariaDB/MySQL (vulnerable SQL query construction).

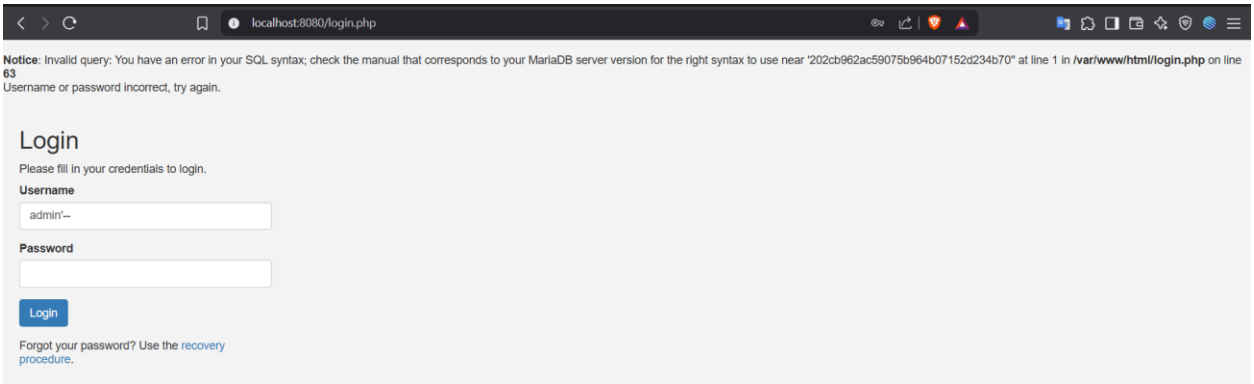
PoC (Proof of Concept)

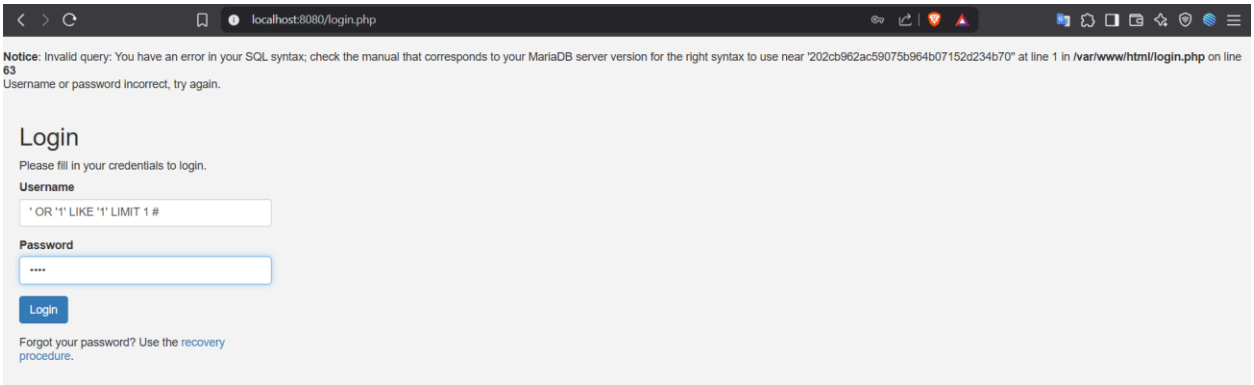
The vulnerability was exploited manually through the login form available at /login.php. By injecting a carefully crafted SQL payload into the username field, the tester was able to manipulate the backend SQL query and gain access without supplying any valid credentials.

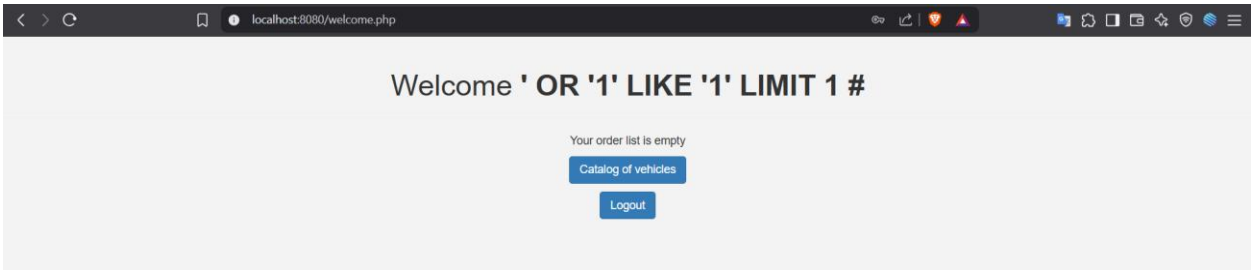
The payload used — ' OR '1' LIKE '1' LIMIT 1 # — breaks out of the intended logic of the SQL statement, introduces a universally true condition, and comments out the rest of the query. This technique bypasses the password check entirely.

Details of the exploit:

- Request URL: http://localhost:8080/login.php.
- Method: POST
- Credentials used:
  - Username: ' OR '1' LIKE '1' LIMIT 1 #
  - Password: test (or any arbitrary string)







Behavior observed:

- The application granted access to the protected area without validating any real credentials.
- No client-side error or warning was displayed.
- The post-login interface appeared, confirming successful authentication bypass.

This confirms that the SQL statement is constructed without parameterization and that user input is embedded directly into the SQL logic.

Impact

Exploitation of this vulnerability results in complete authentication bypass. Any user — even without any prior knowledge of usernames or passwords — can gain access to the application simply by submitting the crafted payload. This compromises the system's confidentiality, integrity, and access controls.

Consequences include:

1. Unauthorized access to restricted areas of the application.
2. Possible privilege escalation if the returned account has admin rights.
3. Loss of data confidentiality and integrity.
4. Increased exposure to further vulnerabilities (XSS, logic flaws, etc.).

Remediation Plan

SQL Injection – Login Bypass

To mitigate this vulnerability, the following steps are strongly recommended:

- **Implement prepared statements** (parameterized queries): Use secure database query methods that treat user input as data, not executable code.
- **Validate and sanitize all user input**: Apply strict server-side validation and input encoding.
- **Disable detailed error messages in production**: Configure the database and application to suppress raw SQL errors from being shown to users.
- **Strengthen authentication logic**: Ensure passwords are hashed using secure algorithms (bcrypt, Argon2) and never rely on plaintext or MD5 hashes.
- **Conduct regular security testing**: Schedule ongoing penetration testing and code reviews to proactively detect injection points and logic flaws.