People's Democratic Republic of Algeria

Ministery of Higher Education and Scientific Research

Ferhat Abbas University of Setif 1



Faculty of Sciences

Computer Science Department

**THEME**

Detecting SQL injection using Deep Learning

Realizedby: Supervised by:

ZITOUNI AHMED FAOUZI DR. BENZINE MEHDI

SEDJAL MOHAMED AYMEN DHIAEDDINE

**2024/2025**

**Table of contenets**

General introduction

Chapter 1 SQL injection

1.1 Introduction

1.2 Web application

1.2.1 Definition…………………………………………………………………………………………………………

1.3 SQL injection………………………………………………………………………………………………………….

1.3.1 Definition6

1.3.2 How SQL Injection Works……………………………………………………….7

1.3.3 Impact of SQL Injection Attacks…………………………………………………

1.4 Techniques of SQL Injection…………………………………………………………….

1.4.1 Tautology-Based SQL Injection…………………………………………………….

1.4.2 Error-Based SQL Injection………………………………………………………….

#### 1.4.3 Blind SQL Injection…………………………………………………………………

**Figures list**

Figure 1.1 Architecture of a Database-driven Web application……………………….

Figure 1.2 SQL Injection attack…………………………………………….

Figure 1.3 Prepared Statements exemple using php………………………….

Figure 1.4 Stored Procedure example using java…………………………………….

Figure 1.5 Input Validation example using java………………………………..

**Chapter 1**

SQL Injections

#### **1.1 Introduction**

With increasingly digital living, web applications are at the core of day to day life from managing finances and online purchasing to collaborating and communicating. This ease of the virtual world comes with inherent security challenges. Cyber attackers persistently evolve their methods to exploit weaknesses, thereby endangering unauthorized data access, downtime of services, and irreparable damage to reputation.

### **1.2 Web applications**

#### **1.2.1 Definition**

Web applications are software programs that operate within a web browser, allowing the user to get a set of interactive functionalities such as login systems for user authentication, and real-time chat or messaging features for immediate communication. By doing so, the user is capable of enjoying advanced and interactive functionalities without the installation of other software. Conversely, the entire code that is required is run in the browser of the web, whereas application logic and data are handled on the server side via the implementation of a Database Management System (DBMS), web applications that use DBMS are known as Database-driven Web applications.

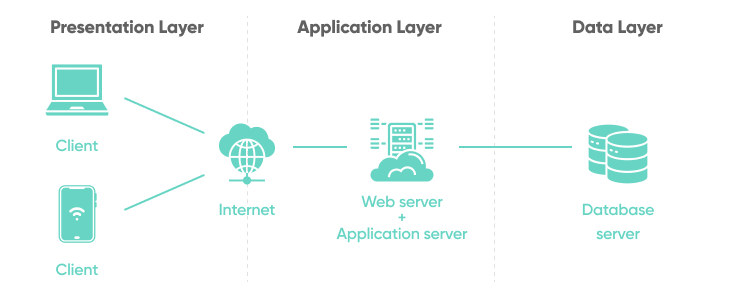
Database-driven Web application are usually composed of a back end database and web pages that include server-side scripts that retrive or update data from the database, depending on user actions like submitting a form, logging in, or searching for products.

Typically, the application relies upon a three-tier structure:

**Presentation Layer:** User interface tier, typically consisting of a web browser or rendering engine using HTML, CSS, and JavaScript.

**Logic Layer:** It handles retrinving and processing the data with the help of server-side scripts.

**Data Layer:** There the data resides, managed by databases including Microsoft SQL Server, MySQL, Oracle, and many more.

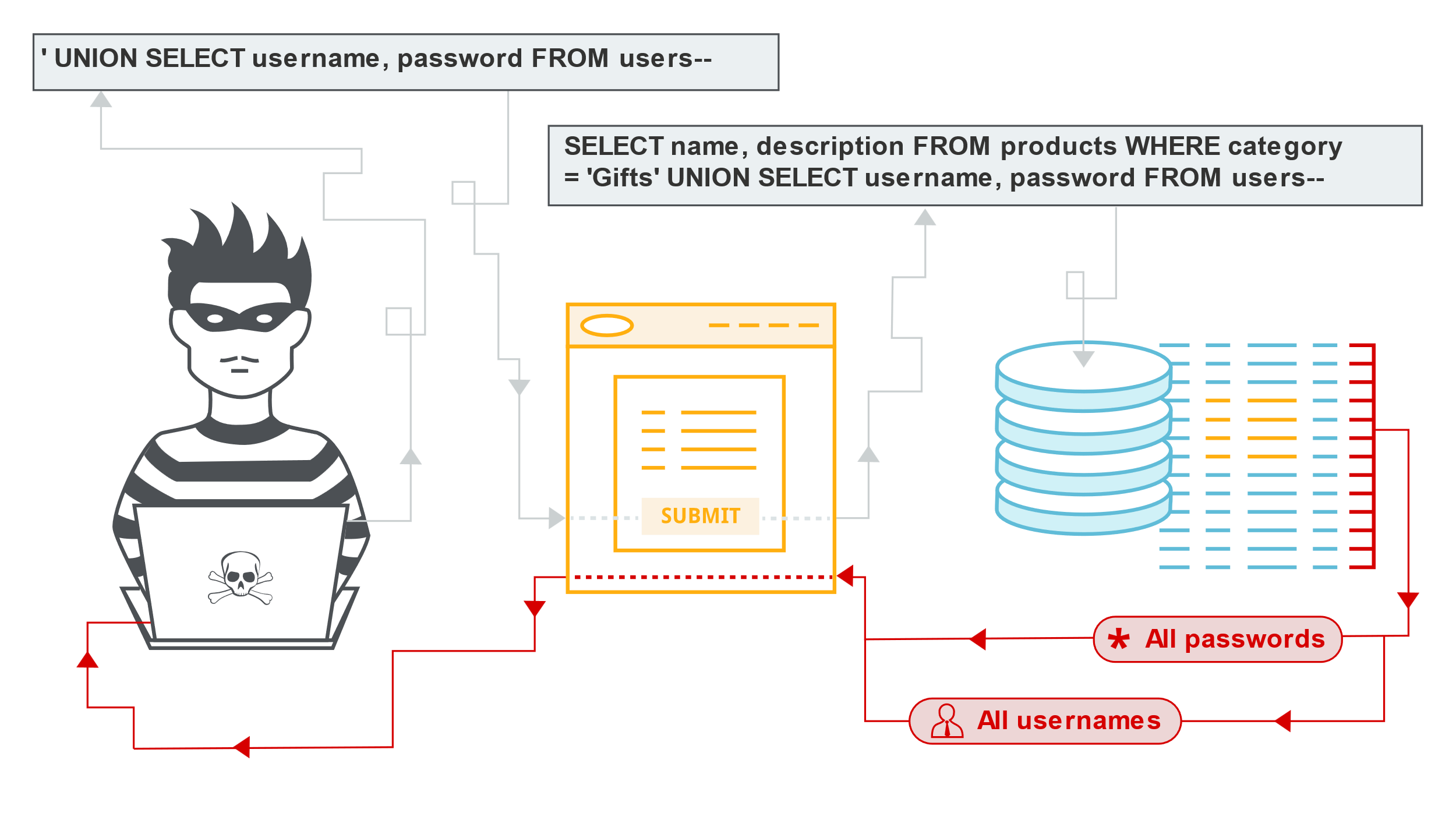


**Figure 1.1** Architecture of a Database-driven Web application

### **1.3 SQL injection**

#### **1.3.1 Definition**

A SQL injection attack consists of insertion or “injection” of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands[[2].](#_[2]__)

**Figure 1.2** SQL Injection attack

**1.3.2 How SQL Injection Works**

It typically involves the following steps:

1. **Identification of vulnerable inputs:** Attackers first identify inputs within the web application that are vulnerable to SQL injection. These inputs could be text fields in a form, URL parameters, or any other input mechanisms.
2. **Crafting the malicious SQL query:** Once a vulnerable input is identified, attackers craft a SQL statement intended to be inserted into the query executed by the application. This statement is designed to modify the original SQL query to perform actions unintended by the application developers.
3. **Bypassing application security measures:** Attackers often have to bypass security measures like input validation or escaping special characters. They achieve this through techniques like string concatenation or utilizing SQL syntax to comment out parts of the original query.
4. **Executing the malicious query:** When the application executes the SQL query, it includes the attacker’s malicious input. This modified query can perform actions such as unauthorized viewing of data, deletion of data, or even database schema alterations.
5. **Extracting or manipulating data:** Depending on the attack, the outcome might be the extraction of sensitive information (like user credentials), altering existing data, adding new data, or even deleting significant portions of the database.
6. **Exploiting database server vulnerabilities:** Advanced SQL injections may exploit vulnerabilities in the database server, extending the attack beyond the database to the server level. This can include executing commands on the operating system or accessing other parts of the server’s file system.

This process leverages the dynamic execution of SQL in applications where user inputs are directly included in SQL statements without proper validation or escaping. It exploits the way SQL queries are constructed, often in a way that the developers did not anticipate[[3].](#_[3]_Bright_security)

##### **Real-Life SQL Injection Attack Examples**

Over the past 20 years, many SQL injection attacks have targeted large websites, business and social media platforms. Some of these attacks led to serious data breaches. A few notable examples are listed below[[3].](SQL#_[3]_OWASP_)

###### **Breaches Enabled by SQL Injection**

* **GhostShell attack**—hackers from APT group Team GhostShell targeted 53 universities using SQL injection, stole and published 36,000 personal records belonging to students, faculty, and staff.
* **Turkish government**—another APT group, RedHack collective, used SQL injection to breach the Turkish government website and erase debt to government agencies.
* **7-Eleven breach**—a team of attackers used SQL injection to penetrate corporate systems at several companies, primarily the 7-Eleven retail chain, stealing 130 million credit card numbers.
* **HBGary breach**—hackers related to the Anonymous activist group used SQL Injection to take down the IT security company’s website. The attack was a response to HBGary CEO publicizing that he had names of Anonymous organization members.

### **1.4 Techniques of SQL Injection**

#### **1.4.1 Error-Based SQL Injection**

**Error-based SQL injection** is a catastrophic security vulnerability where attackers introduce malicious SQL statements to a web application's input fields. Exploiting it makes the app spit out SQL errors that consequently reveal database schema's sensitive data, contents, or system settings inadvertently.

**How It Works:**

The attack begins when an attacker finds an injectable input field, i.e., login page or search box, in which user input is inserted within SQL queries. The attacker inserts specially designed SQL code, resulting in SQL syntax errors because the query is executed. The syntax errors provide the attacker with database information, which enables them to steal or tamper with data.

**Example**[[4]](#_[4]_Dafydd_Stittard)**: Conditional Errors in Oracle/MS-SQL**

|  |
| --- |
| SELECT 1/0 FROM dual  WHERE (SELECT username FROM all\_users WHERE username = 'DBSNMP') = 'DBSNMP'; |

**Explanation**:

1. **Subquery**:
   * The subquery (SELECT username FROM all\_users WHERE username = 'DBSNMP') checks if a user named DBSNMP exists in the all\_users table.
2. **Condition**:
   * If the user DBSNMP exists, the condition (SELECT username FROM all\_users WHERE username = 'DBSNMP') = 'DBSNMP' evaluates to TRUE.
3. **Error Induction**:
   * When the condition is TRUE, the database evaluates the expression 1/0, which causes a **divide-by-zero error**.
   * If the condition is FALSE (i.e., the user does not exist), the expression 1/0 is **not evaluated**, and no error occurs.

**Detection**:

* If the application returns an **HTTP 500 error** or a database error message, the attacker can infer that the condition is TRUE (i.e., the user DBSNMP exists).
* If no error occurs, the condition is FALSE (i.e., the user does not exist).

**Advanced Use Case: Data Exfiltration**

**Scenario**:

* A web application allows users to sort search results using a sort parameter:

|  |
| --- |
| /search.jsp?department=30&sort=ename |

The backend SQL query:

|  |
| --- |
| SELECT ename, job, deptno, hiredate FROM emp  WHERE deptno = ?  ORDER BY [param\_sort] DESC; |

**Malicious Injection**:

The attacker injects a payload into the sort parameter to test a condition:

|  |
| --- |
| /search.jsp?department=20&sort=(SELECT 1/0 FROM dual  WHERE (SELECT SUBSTR(MAX(object\_name),1,1) FROM user\_objects)='Y'); |

**Explanation**:

1. **Subquery**:
   * The subquery (SELECT SUBSTR(MAX(object\_name),1,1) FROM user\_objects) extracts the **first character** of the largest object name in the user\_objects table.
2. **Condition**:
   * If the first character is 'Y', the condition evaluates to TRUE, and the database attempts to evaluate 1/0, causing a **divide-by-zero error**.
   * If the first character is not 'Y', no error occurs, and the query returns results normally.
3. **Inference**:
   * The attacker can use this technique to **brute-force** each character of the object name by testing different values (e.g., 'A', 'B', 'C', etc.).

**1.4.2 Blind SQL Injection**

Blind SQL (Structured Query Language) injection is a type of [SQL Injection](https://owasp.org/www-community/attacks/SQL_Injection) attack that asks the database true or false questions and determines the answer based on the applications response. This attack is often used when the web application is configured to show generic error messages, but has not mitigated the code that is vulnerable to SQL injection.

When an attacker exploits SQL injection, sometimes the web application displays error messages from the database complaining that the SQL Query’s syntax is incorrect. Blind SQL injection is nearly identical to normal [SQL Injection](https://owasp.org/www-community/attacks/SQL_Injection), the only difference being the way the data is retrieved from the database. When the database does not output data to the web page, an attacker is forced to steal data by asking the database a series of true or false questions. This makes exploiting the SQL Injection vulnerability more difficult, but not impossible [[5].](Blind#_[5]_OWASP,_)

##### **1.4.2.1 Content-Based Blind SQL Injection**

**How It Works:**  
 Unlike traditional SQL injection, where database error messages expose data directly, blind SQL injection does not return query results to the user. Attackers exploit this by providing conditional queries and observing the application's response to infer data from the database.

**Technical Explanation:**  
 Blind SQL injection relies on evaluating conditions based on the application's responses. Attackers inject SQL conditions and observe response differences (e.g., message change or page behavior) to deduce data.

**Example:**

vulnerable web application allows the attacker to inject SQL on a URL parameter that fetches data based on an id. The attacker confirms the vulnerability by injecting true/false statements in the id parameter to identify between valid and invalid SQL queries.

The attacker first sends a request like:

|  |
| --- |
| http://newspaper.com/items.php?id=2 |

This executes:

|  |
| --- |
| SELECT title, description, body FROM items WHERE ID = 2; |

Next, the attacker tests for SQL injection by adding a false condition:

|  |
| --- |
| http://newspaper.com/items.php?id=2 and 1=2 |

The query becomes:

|  |
| --- |
| SELECT title, description, body FROM items WHERE ID = 2 AND 1 = 2; |

Since 1=2 is false, the page returns no content, confirming the injection.

Then, the attacker tests a true condition:

|  |
| --- |
| http://newspaper.com/items.php?id=2 and 1=1 |

The query becomes:

|  |
| --- |
| SELECT title, description, body FROM items WHERE ID = 2 AND 1 = 1; |

This question retrieves the anticipated data, revealing the vulnerability.Contrasting the output of these two injections, the attacker can determine that the page is vulnerable to SQL injection and proceed to pull data from the database[[5]](Blind#_[5]_OWASP,_).

**Explanation:**

Here, the attacker uses a spurious condition (and 1=2) to decide whether the page is vulnerable to SQL injection. Since no information is returned, the attacker confirms the vulnerability. A real condition (and 1=1) provides expected information, ascertaining that the injection has been successful. The attacker now iterates data, e.g., table names or other confidential data, using similar true/false conditions based on the database schema.

##### **1.4.3.2 Time-Based Blind SQL Injection**

**How It Works:**  
 In time-based blind SQL injection, attackers use SQL functions like **SLEEP()** to introduce a delay in the server’s response. If the delay occurs, it indicates the injected SQL condition is true; if not, it is false. This helps attackers extract data even when no visible content is returned.

**Technical Explanation:**  
The attacker sends queries that include conditional delays, such as:  
xyz' AND IF(1=1, SLEEP(5), 0) – The server delays for 5 seconds, confirming the condition is true.  
xyz' AND IF(1=2, SLEEP(5), 0) – No delay occurs, confirming the condition is false.

**Example:**

Consider a web application that retrieves user information based on a user ID provided in the URL:

|  |
| --- |
| http://example.com/user.php?id=1 |

The corresponding SQL query might be:

|  |
| --- |
| SELECT \* FROM users WHERE id = 1; |

If the application is vulnerable to Time-Based Blind SQL Injection, an attacker can manipulate the id parameter to include a time delay function.

**Malicious Injection:**

|  |
| --- |
| http://example.com/user.php?id=1; IF(1=1, SLEEP(5), 0); |

In this example, the injected SQL statement includes a conditional function that causes the database to pause for 5 seconds if the condition 1=1 is true.

**Explanation:**

Here, the attacker is inserting a conditional SQL function that will intentionally delay the database's response time. The inserted SQL query is:

|  |
| --- |
| SELECT \* FROM users WHERE id = 1; IF(1=1, SLEEP(5), 0); |

The IF(1=1, SLEEP(5), 0) function will evaluate the condition 1=1, which is always fulfilled. So the SLEEP(5) function will be invoked, and the database will take 5 seconds to reply.

If the application responds in 5 seconds, the attacker confirms successful injection and exposure of the application to Time-Based Blind SQL Injection.

The attackers may use this technique to provide educated guesses on the structure and content of the database even when direct extraction is not possible.

### **1.4.3 Tautology-Based SQL Injection**

The term ‘tautology’ originates from the field of logic, where it is used to describe a statement that is always true, regardless of the truth values of its components. In other words, a tautological statement is one that is true by virtue of its logical form alone [[6]](#_[6]_Moxso,_“SQL).

**How It Works:**

This attack exploits the use of tautological SQL statements always result in true and thus bypasses authentication and other security measures.

**Technical Explanation:**  
 A tautology is a logical statement that remains true under any combination of values. Malicious users insert such statements into SQL queries, compelling the database to execute and authenticate unauthorized requests.

|  |
| --- |
| SELECT \* FROM users WHERE username = ‘admin’OR ‘1’=’1’ – AND password =’anything’; |

**OR '1'='1'** in this example guarantees the condition will always be true, giving access.  
Real-World Scenario: An attacker hacks a login form on a web page, bypassing user authentication and accessing an administrator account**.**

### **1.5 Methods to prevent SQL Injection attacks.**

Attackers can use SQL injection on an application if it has dynamic database queries that use string concatenation and user supplied input. To avoid SQL injection flaws.

There are simple techniques for preventing SQL injection vulnerabilities and they can be used with practically any kind of programming language and any type of database [[4]](#_[4]_OWASP_Cheat_1).

#### **1.5.1 Prepared Statements (with Parameterized Queries)**

When developers are taught how to write database queries, they should be told to use prepared statements with variable binding (also known as parameterized queries). Prepared statements are simple to write and easier to understand than dynamic queries, and parameterized queries force the developer to define all SQL code first and pass in each parameter to the query later.

If database queries use this coding style, the database will always distinguish between code and data, regardless of what user input is supplied. Also, prepared statements ensure that an attacker cannot change the intent of a query, even if SQL commands are inserted by an attacker.

In PHP, PHP Data Objects (PDO) offer a more effective approach to database interactions. By providing methods that simplify parameterized queries, PDO ensures that user input is always treated as data rather than executable SQL code and enhances code readability and also ensures greater portability across multiple databases[.](#_[4]_OWASP_Cheat)

**Figure 1.3** Prepared Statements example using php[[4]](#_[4]_OWASP_Cheat)

< ?php

$dbh = new PDO('mysql:dbname=testdb;host=127.0.0.1', $user, $password);

$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (:name,:value)");

$stmt->bindParam(':name', $name);

$stmt->bindParam(':value', $value);

$stmt->execute();

#### **1.5.2 Stored Procedures**

Though stored procedures are not always safe from SQL injection, developers can use certain standard stored procedure programming constructs. This approach has the same effect as using parameterized queries, as long as the stored procedures are implemented safely (which is the norm for most stored procedure languages).

**Safe Approach to Stored Procedures :**

If stored procedures are needed, the safest approach to using them requires the developer to build SQL statements with parameters that are automatically parameterized, unless the developer does something largely out of the norm. The difference between prepared statements and stored procedures is that the SQL code for a stored procedure is defined and stored in the database itself, then called from the application.

Since prepared statements and safe stored procedures are equally effective in preventing SQL injection, your organization should choose the approach that makes the most sense for you.

The following code example uses Java’s implementation of the stored procedure interface (CallableStatement) to execute the same database query.

// This should REALLY be validated

String custname = request.getParameter("customerName");

try {

  CallableStatement cs = connection.prepareCall( "{call sp\_getAccountBalance(?)}" );

  cs.setString(1, custname);

  ResultSet results = cs.executeQuery();

  // … result set handling

} catch (SQLException se) {

  // … logging and error handling

}

**Figure 1.4** Stored Procedure example using java

#### **1.5.3 Input Validation**

Input validation is performed to ensure only properly formed data is entering the workflow in an information system, preventing malformed data from persisting in the database and triggering malfunction of various downstream components.

Input validation should happen as early as possible in the data flow, preferably as soon as the data is received from the external party.

Data from all potentially untrusted sources should be subject to input validation, including not only Internet-facing web clients but also backend feeds over extranets, from [suppliers, partners, vendors or regulators](https://badcyber.com/several-polish-banks-hacked-information-stolen-by-unknown-attackers/), each of which may be compromised on their own and start sending malformed data.

Example validating the parameter "zip" using a regular expression :

private static final Pattern zipPattern = Pattern.compile("^\d{5}(-\d{4})?$");

public void doPost( HttpServletRequest request, HttpServletResponse response) {

try {

      String zipCode = request.getParameter( "zip" );

      if ( !zipPattern.matcher( zipCode ).matches() ) {

          throw new YourValidationException( "Improper zipcode format." );

      }

      // do what you want here, after its been validated ..

  } catch(YourValidationException e ) {

      response.sendError( response.SC\_BAD\_REQUEST, e.getMessage() );

  }

}

**Figure 1.5** Input validation example using java[[5]](#_[5]_OWASP_Cheat)

#### **1.5.4 Escaping All User-Supplied Input**

In this approach, the developer will escape all user input before putting it in a query. It is very database specific in its implementation. This methodology is frail compared to other defenses, and we CANNOT guarantee that this option will prevent all SQL injections in all situations.

If an application is built from scratch or requires low risk tolerance, it should be built or re-written using parameterized queries, stored procedures, or some kind of Object Relational Mapper (ORM) that builds your queries for you.

**1.6 Conclusion**

SQL injection continues to pose a significant threat to web applications, even though effective countermeasures are available, such as input validation, use of prepared statements, and escaping the user input.

However, due to the large variety of SQL injection attacks, it tends to fail in protecting the sensitive data in the databases. For that reason, it is recommended to apply the techniques mentioned before.

#### **References**

#### [1]

#### [2] OWASP “SQL Injection (SQLI)”:

[https://owasp.org/search/?searchString=sql+injection+](%20https://owasp.org/search/?searchString=sql+injection+)

#### [3] Bright security “SQL Injection Attack”:

<https://brightsec.com/blog/sql-injection-attack/>

#### [4] Dafydd Stittard and Marcos Pinto,The web Application Hacker’s Handbook: Finding and Exploiting Security Flaws, 2nd Edition, Wiley Publishing Inc, 2011.

#### [5] OWASP, “Blind SQL injection”:

<https://owasp.org/www-community/attacks/Blind_SQL_Injection>

#### [6] Moxso, “SQL Tautology” :

<https://moxso.com/blog/glossary/tautology>

#### 

#### [] OWASP Cheat Sheet Series, “SQL Injection Prevention”:

<https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html>

#### [] OWASP Cheat Sheet Series, “Query Parameterization”:

<https://cheatsheetseries.owasp.org/cheatsheets/Query_Parameterization_Cheat_Sheet.html>

#### [] OWASP Cheat Sheet Series, “Input Validation”:

<https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html>