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…………………………………………………………………

**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.

        Instead of going with the traditional enumeration, it is pertinent to talk about these five new categories of risks that contemporary web applications have to face:

**Inadequate Input Validation:**

Not validating and sanitizing user-supplied data properly can facilitate a range of attacks, such as injection and cross-site scripting (XSS). Such neglect permits attackers to inject malicious code into the system, thereby violating data integrity and application functionality.

**Insufficient Authentication and Session Management:**

Insufficient methods of validating user identities or handling active sessions can permit unauthorized users to obtain access. Adequate controls such as multi-factor authentication and secure session management are essential for protecting user accounts and sensitive information.

**Lack of data protection:**

Irrespective of whether the data is at rest or in motion, poor encryption or weak key management practices can lead to exposure of sensitive data. Robust cryptographic standards have to be implemented, and sensitive data has to be maintained confidentially from its life cycle for successful protection of data.

**Defective Integration & Dependency Management:**

Contemporary web applications tend to include external services and libraries. When these underlying elements are insecure or outdated, they can inject vulnerabilities into the system overall. Diligent monitoring ofdependencies, along with routine security audits, is required to avert these threats.

**Poor error handling and logging:**

Poorly controlled error messages and a lack of logging can inadvertently disclose system information to prospective attackers or hamper the timely detection of a security incident. Having well defined error handling routines and utilizing solid monitoring systems are essential for the timely detection and containment of malicious activity.

One of the most risky implications of these vulnerabilities is exposure to SQL injection attacks. Where an application does not sanitize input sufficiently, malicious users can inject malicious SQL statements, and these have the potential to result in unauthorized data alteration or even system hijacking. The implication of this type of exploitation can be catastrophic resulting in data loss, system crash, and long-term reputational damage for an organization.

        Through the use of proactive design principles and audits on a regular basis, developers can greatly minimize the chances of exploitation in today's intricate digital world.

### **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.

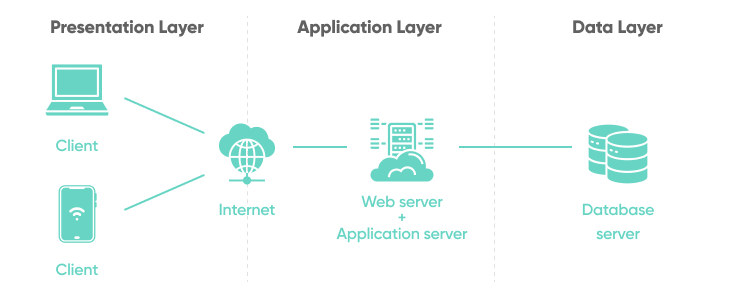
Database-driven Web application are usually composed of a back end database and web pages that include server-side scripts that retrive or update data from the database, depending on user actions like submitting a form, logging in, or searching for products.

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.



### **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]_OWASP_“SQL)

**1.3.2 How SQL Injection Works**

SQL injection is a type of cyberattack which exploits vulnerabilities within the input validation process of an internet application. It occurs when a malicious SQL code is inserted or placed through user inputs, which are later executed by a database. The reason being the application fails to validate, sanitize, or parameterize user input appropriately prior to their inclusion within SQL queries.

**Mechanism of SQL Injection**

**1 User Input Handling :**

Web applications are prone to accept user inputs (e.g., from login forms, search fields, or URL parameters) and use them to construct SQL queries dynamically. For example:



**2 Malicious Input Injection :**If the application does not sanitize or validate the input, an attacker can inject a malicious SQL code. For example, entering ' OR '1'='1 as the username and an empty password transforms the query into:



The condition **'1'='1'** is always true, causing the query to return all rows in the users table. This allows the attacker to bypass authentication.

**3 Database Execution** :  
The database executes the malicious query, leading to unauthorized access, data leakage, or other malicious activity.

**1.3.3 Impact of SQL Injection Attacks**

SQL injection (SQLi) is an attack that takes advantage of vulnerabilities in the application's software by injecting malicious SQL statements into form fields. The consequences of SQL injection are severe and include:

**1. Data Theft**: Sensitive information from the database, such as personal data, credentials, and financial information, can be stolen by attackers.

**2. Modification of Data**: SQL injection can enable attackers to modify or delete data, resulting in integrity issues and loss of critical information.

**3. Bypassing Authentication**: SQL query attacks can enable unauthorized access to users' accounts or administrative rights.

**4. Denial of Service (DoS):** Attackers can generate queries that overwhelm the database, which will subsequently fail or become unavailable.

**5. Remote Code Execution**: In some cases, SQL injection is employed to run unauthorized code on the server, taking full control of the system by attackers.

##### ***6. Reputation Damage****: Organizations that fall victim to SQL injection attacks risk losing business reputation, customers' trust, and possible legal repercussions, most significantly if it involves sensitive information.*

##### ***7. Loss of Funds****: The expense of data breaches, including recovery, compliance penalties, and lost business, can be significant.*

##### *Briefly, SQL injection is a security threat that demonstrates the value of good security controls, such as input validation, parameterized queries, and security auditing.*

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

**1.4.1 Tautology-Based SQL Injection**

This method exploits the use of conditional SQL statements that always return true, effectively bypassing authentication and other access controls.

A tautology is a logical statement that is always true, regardless of its components. Attackers inject such statements into SQL queries, which forces the database to process and accept unauthorized requests.



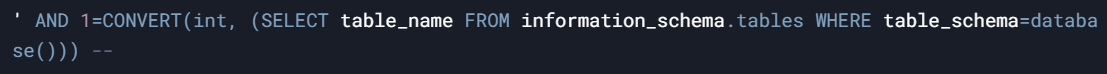
* Here, OR '1'='1' ensures the condition is always true, granting access.
* **Real-World Scenario:** An attacker targets a login form on a website, bypassing user authentication and accessing an administrator account.
* **Impact:**
  + Unauthorized access to accounts.
  + Potential data breaches and manipulation.
  + Loss of user trust and legal consequences.
* **Countermeasures:**
  + Employ parameterized queries.
  + Sanitize user inputs rigorously

.

**1.4.2 Error-Based SQL Injection**

This technique relies on triggering and extracting database error messages to obtain information about the database structure.

By deliberately generating SQL errors, attackers can gather information such as table structures, column types, and even data.



* **Real-World Scenario:** A hacker attacks a poorly configured e-commerce site, revealing table names containing customer payment details.
* **Impact:**
  + Exposure of database schema.
  + Facilitation of more precise attacks.
  + Leakage of sensitive data.
* **Countermeasures:**
  + Disable detailed error reporting in production.
  + Implement comprehensive error handling.

##### **1.4.3 Blind SQL Injection**

This technique is used when the application does not display database errors, forcing attackers to infer database behavior indirectly.

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

* **Detailed Explanation:** Attackers use conditional statements to observe subtle changes in the application's content or behavior.



* **Impact:**
  + Gradual extraction of data through logical inference.
  + Highly challenging to detect.
* **Countermeasures:**
  + Use of web application firewalls (WAFs).
  + Regular code reviews.
* Ssss

### **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[[3]](#_[3]_OWASP_\“SQL).

#### **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.

$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

}

#### **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() );

  }

}

#### **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.

#### **References**

#### [1] “

#### [2] OWASP “SQL Injection (SQLI)”: [https://owasp.org/search/?searchString=sql+injection+]( https://owasp.org/search/?searchString=sql+injection+)

#### [3] OWASP “SQL Injection Prevention“ : <https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html>

#### 