Reference

Definition **web app (**[**1**](brouillon%20aymen.docx)**)**

<https://www.techtarget.com/searchsoftwarequality/definition/Web-application-Web-app>

)

A web application (web app) is an [application program](https://www.techtarget.com/searchsoftwarequality/definition/application) that is stored on a remote server and delivered over the internet through a browser interface. [Web services](https://www.techtarget.com/searchapparchitecture/definition/Web-services) are web apps by definition and many, although not all, websites contain web apps.

Developers [design web applications](https://www.techtarget.com/searchcloudcomputing/definition/web-application-development) for a wide variety of uses and users, from an organization to an individual for numerous reasons. Commonly used web applications can include webmail, online calculators, [social networking](https://www.techtarget.com/whatis/definition/social-networking) or [e-commerce](https://www.techtarget.com/searchcio/definition/e-commerce) shops. While users can only access some web apps by a specific browser, most are available no matter the browser.

Unlike traditional desktop applications that are installed on a computer, web apps can be used on any device with a web browser, including smartphones, tablets and desktops.

**Ou (**[**https://www.sciencedirect.com/topics/computer-science/web-application**](https://www.sciencedirect.com/topics/computer-science/web-application)**)**

**-**

**- PAS finis**

**--**

**How sql works**

* 1.3.2 How SQL Injection Works

A **SQL Injection attack** involves **inserting** or “**injecting**” a SQL query via the input data from the client to the application. A successful attack allows an attacker to manipulate the SQL queries that an application makes to its database. 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.

([**https://brightsec.com/blog/sql-injection-attack/**](https://brightsec.com/blog/sql-injection-attack/)**) (1)**

**Real EXEMPLES**

## 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.

### 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.

**Meme source + dok nzidoou figure te3 exemple**

([**https://brightsec.com/blog/sql-injection-attack/**](https://brightsec.com/blog/sql-injection-attack/)**) (1)**

**Finis**

**Impact**

**1.3.3 Impact of SQL Injection Attacks**

With no mitigating controls, SQL injection can leave the application at a high risk of compromise resulting in an impact on the **confidentiality**, and **integrity** of data as well as **authentication** and **authorization** aspects of the application. An adversary can steal sensitive information stored in databases used by vulnerable programs or applications such as user credentials, trade secrets, or transaction records.

SQL injection vulnerabilities should *never be left open*; they **must** be fixed in all circumstances. If the authentication or authorization aspects of an application is affected an attacker may be able login as any other user, such as an administrator which elevates their privileges.

**(**[**https://www.packetlabs.net/posts/how-does-sql-injection-impact-clients/**](https://www.packetlabs.net/posts/how-does-sql-injection-impact-clients/)**) (2)**

**The consequences of a successful SQL injection attack can include:**

* **Stolen credentials—**attackers can obtain credentials via SQLi and then impersonate users and use their privileges.
* **Unauthorized access to databases—**attackers can gain access to the sensitive data in database servers.
* **Data alteration—**attackers can alter or add new data to the accessed database.
* **Data deletion—**attackers can delete database records or drop entire tables.

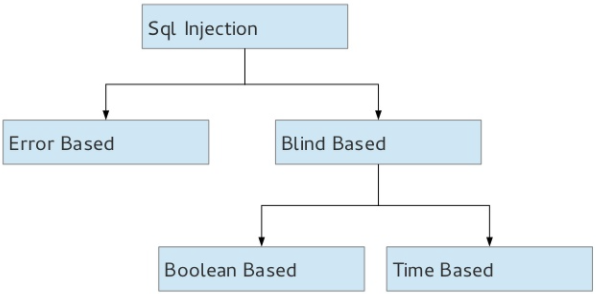
**Lateral movement—**attackers can access database servers with operating system privileges, and use these permissions to access other sensitive systems.

**(**[**https://brightsec.com/blog/sql-injection-attack/**](https://brightsec.com/blog/sql-injection-attack/)**) (1)**

**finis**

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

Figure (3) (https://www.greycampus.com/opencampus/ethical-hacking/types-of-sql-injection)



**1.4.1 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. ( 4)( <https://moxso.com/blog/glossary/tautology>)

**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**.**

**Impact:**  
Unauthorized account access.  
Data breaches and manipulation potential.  
Loss of user trust and legal consequences.

**1.4.2 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: 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.).

(10)( The Web Application Hacker’s Handbook page 321) pour l exemple

**Technical Explanation**

Error-based SQL injection is an In-band injection attack, whereby the same channel that is used to attack is also utilized to plunder database data. It ranks among the most used and simplest intrusion techniques due to its effectiveness and simplicity.(6)

( ta9der ttbdl ntal3ouha f blsat how itswork )

Attackers exploit vulnerabilities to cause the **database** to return error messages instead of the intended data. The error messages have a tendency to disclose sensitive database schema, table structure, and even stored data, which makes it easy for attackers to plan subsequent attacks.

**(6)(** [**https://beaglesecurity.com/blog/vulnerability/error-based-sqli.html**](https://beaglesecurity.com/blog/vulnerability/error-based-sqli.html)**)**

**Impacts**

**Error-based SQL injection** renders the web applications along with the back-end databases highly susceptible. Depending on the scale of the vulnerability, along with the intent and capability of the attacker, the impact may fluctuate. The following are some of the impactful effects of error-based SQL injection attacks:

**Unauthorized Data Access**:

Attackers can view sensitive data stored in the database such as user accounts, personal data, financial data, and intellectual property, thus violating data confidentiality.

**Data Manipulation**:

In addition to information retrieval, attackers also destroy or delete database data, leading to data loss or unauthorized alteration resulting in data corruption or compromise in data integrity and application functionality.

**Data Disclosure:**

Attackers' error messages to the database consist of structural information such as table names and column names, etc. Structural information such as this is useful for the attacker to plan and orchestrate more elaborate attacks.

**Information Leakage**:

Error messages that are produced due to application crashes may expose information related to the codebase in a way that is identifiable and exploitable by the attackers as well.

**Application Disruption:**

SQL injection attacks repeatedly executed can flood the database server, leading to Denial of Service (DoS) attacks that render the application unavailable to authorized users.

**Regulatory Compliance Violations**:

SQL injection data breaches can lead to a violation of data protection legislation. Organizations can attract lawsuits, penalties, and compliance issues, particularly where sensitive data is involved.

**Financial Loss:**

The price of a successful SQL injection attack encompasses security remediation efforts, legal fees, user compensation affected by the attack, and implementing stronger security controls.

**Loss of Customer Trust**:

Security breaches can ruin an organization's reputation forever. Users lose business and integrity as they start suspecting the system guarding their data.

Sophisticated hackers will leverage error-based SQL injection as a foothold to persistently probe and attack the rest of the network, escalating privileges and breaching more sensitive systems.

9adrin na9sou fihom ( impact )

#### 1.4.3 Blind SQL Injection

**Blind SQL injection** is another type of SQL injection attack where the attacker presents the database with true or false questions and makes their answers rely on answers from the application. It is often used where the web application provides generic error messages but still has code that is vulnerable to SQL injection. (8)

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

(**11**) <https://owasp.org/www-community/attacks/Blind_SQL_Injection>

**Scenario:**

 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.

**Malicious Injection:**  
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.

**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.

**Scenario:**

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.(**11)**

**Impact:**

* **Unauthorized Data Access:** Sensitive information like user credentials can be retrieved.
* **Data Manipulation:** The intruders can insert, delete, or modify data.
* **Application Disruption:** Different shots can lead to issues with system functionality.
* **Legal Sanctions:** Failure to comply may result in breach of data protection law.
* **Loss of Trust:** Individuals lose confidence in the security of the system

(7)( <https://portswigger.net/web-security/sql-injection/blind>)

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

**1.4.4 Union-Based SQL Injection**

**How It Works:**

The UNION operator is used in SQL to combine the results of two or more SELECT statements into a single result set. When a web application contains a SQL injection vulnerability that occurs in a SELECT statement(**9**), attackers can utilize this operator to insert an additional query and merge its outcome with the outcome of the initial query.

**Technical Explanation:**

Using this method, malicious users can retrieve unauthorized data from the database. UNION-based SQL injection is widely supported by all the major database management systems (DBMS) and is generally the best way to extract specific database contents when query results are directly presented on the application interface.

### ****Example 1: Extracting the Current Database User****

#### ****Scenario (12)****

A web application shows product information based on a product ID passed in the URL. The application is susceptible to SQL injection since it puts user input directly into the SQL query without sanitizing. The attacker finds such vulnerability and chooses to exploit it in order to get the current database user, which can assist them in knowing the access level they have and strategize future attacks.  
  
The attacker knows that the application has a database backend (for example, Microsoft SQL Server, MySQL, or Oracle) and wishes to extract the username of the account which issued the queries. .

#### ****Malicious Injection:****

The attacker sends the following malicious URL to the application:

|  |
| --- |
| http://www.victim.com/products.asp?id=12+union+select+NULL,system\_user,NULL,NULL |

#### ****Explanation:****

1. **Original Query:**  
   The application executes the following query to retrieve product details:

|  |
| --- |
| SELECT id, type, description, price FROM products WHERE id = 12 |

This query returns the details of the product with ID 12.

1. **Injected Query:**  
   The attacker appends a UNION SELECT statement to the original query to retrieve the current database user:

|  |
| --- |
| UNION SELECT NULL, system\_user, NULL, NULL |

* + The UNION operator combines the results of the original query with the results of the injected query.
  + The system\_user function (or equivalent, depending on the database) retrieves the username of the current database user.
  + The NULL values are used to match the number of columns in the original query (since the injected query only needs one column for the username, but the original query returns four columns).

1. **Combined Query:**  
   The database executes the following combined query:

|  |
| --- |
| SELECT id, type, description, price FROM products WHERE id = 12  UNION  SELECT NULL, system\_user, NULL, NULL |

#### ****Result:****

|  |  |  |  |
| --- | --- | --- | --- |
| **Id** | **Type** | **Description** | **Price** |
| 12 | Book | SQL Injection Attacks | 50 |
| NULL | db\_user | NULL | NULL |

### ****Example 2: Extracting Multiple Rows from the****customers****Table****

#### ****Scenario: (12)****

A web application displays product details based on a product ID passed in the URL. The application is vulnerable to SQL injection because it directly incorporates user input into the SQL query without proper sanitization. The attacker discovers this vulnerability and decides to exploit it to extract sensitive customer data from the customers table in the database.

The attacker's goal is to retrieve the full list of customers (first and last names) from the database.

#### ****Malicious Injection:****

The attacker sends the following malicious URL to the application:

|  |
| --- |
| http://www.victim.com/products.asp?id=12+union+select+userid,first\_name,second\_name,NULL+from+customers |

#### ****Explanation:****

1. **Original Query:**  
   The application executes the following query to retrieve product details:

|  |
| --- |
| SELECT id, type, description, price FROM products WHERE id = 12 |

This query returns the details of the product with ID 12.

1. **Injected Query:**  
   The attacker appends a UNION SELECT statement to the original query to retrieve data from the customers table:

|  |
| --- |
| UNION SELECT userid, first\_name, second\_name, NULL FROM customers |

* + The UNION operator combines the results of the original query with the results of the injected query.
  + The NULL value is used to match the number of columns in the original query (since the customers table has only three columns, but the original query returns four columns).

1. **Combined Query:**  
   The database executes the following combined query:

|  |
| --- |
| SELECT id, type, description, price FROM products WHERE id = 12  UNION  SELECT userid, first\_name, second\_name, NULL FROM customers |

**Result: (12)**

|  |  |  |  |
| --- | --- | --- | --- |
| id | **Type** | **Description** | **Price** |
| 12 | Book | SQL Injection Attacks | 50 |
| 1 | aymen | sed | NULL |
| 2 | aymen | sed | NULL |
| 3 | aymen | sed | NULL |

(9) (**Dafydd, Stittard Marcos Pinto, The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws, 2nd Edition, Wiley Publishing Inc, 2011**.)

(12) SQL inection attacks and deffemce SYNGRESS livre page 174

**1.4.1 Tautology-Based SQL Injection**

**This attacks depend upon taking advantage of the employment of conditional SQL always True, therefore circumventing authentication and access control.**

**Tautology refers to a statement, no matter parts, which is always true.**

**Attackers sow such statements within SQL queries for injecting into databases, therefore, the database continues to process unauthorized requests.**

**•\tOR '1'='1' is employed here such that the condition will always be true, finding entry.**

**•\tReal-World Scenario: A hacker compromises a login form of a website and bypasses user authentication to utilize an administrator's account.**

**•\tConsequence:**

**o\textbf{Unintended access of accounts.}**

**o\textbf{Possible breaches and tampering of data.}**

**o\textbf{Damage to user confidence and legal recourse.}**

**•\tPreventive measures:**

**o\textbf{Use parameterized queries.}**

**o\textbf{Sanitize the user input string thoroughly.}**

**1.4.2 Error-Based SQL Injection**

**This technique utilizes provoking and retrieving database error messages to learn information about the database schema.**

**By deliberately generating SQL errors, attackers can extract information such as table structure, column types, and even data.**

**•\tReal-World Scenario: An ill-configured e-commerce website is hacked by a hacker, where table names containing customer payment details are revealed.**

**•\tImpact:**

**o\tdisclosure of database schema.**

**o\tenabling more precise attacks.**

**o\tdisclosure of sensitive data.**

**•\tCountermeasures:**

**o\tdisable detailed error reporting in production.**

**o\tenforce strict error handling.**

**1.4.3 Blind SQL Injection**

**This technique is used when the application does not return database error messages and attackers have to make an indirect inference about database behavior.**

**1.4.3.1 Content-Based Blind SQL Injection**

**•\\tDetailed Description: Attackers make use of conditional statements to find minute differences in the application content or behavior.**

**•\\tImpact:**

**o\\tStealing data bit by bit through logical inference.**

**o\\tExtremely hard to find.**

**•\\tCountermeasures:**

**o\\tUse of web application firewalls (WAFs).**

**o Regular code reviews.**