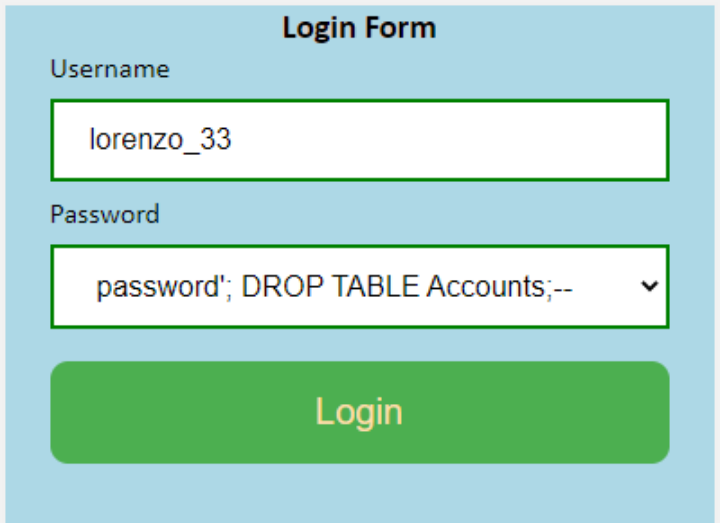


Preventing SQL Injection Attacks

SQL Injection

A SQL injection is a serious vulnerability affecting applications that use SQL as their database language. Through cleverly constructed text inputs that modify the backend SQL query, threat actors can force the application to output private data or respond in ways that provide intel. SQL injections attacks can ultimately be used to steal information and even take complete control of a system.



Login Form

Username
lorenzo_33

Password
password'; DROP TABLE Accounts;--

Login

Types of SQL Injections

SQL injections can be broken into multiple types depending on how information is retrieved: union-based, error-based, boolean-based, time-based, and out-of-band injections.

Mitigating SQL Injection Attacks: Input Sanitization

One way SQL injections can be mitigated is through input sanitization. Sanitization is the process of removing dangerous characters from user input. Dangerous characters might include:

,
;
\--

This is important because they allow attackers to extend SQL queries to gain more information from a database. Careful, this method is not the perfect defense against SQL injections. Removing characters may have no effect in some queries and, if an attacker finds a way to bypass the sanitization process, they can easily inject data into your system.

```
SELECT username, email FROM users WHERE id  
= '1' AND '1' = '2';
```

Union-Based Injections

Uses the `UNION` SQL keyword to take two separate `SELECT` queries and combine their results. Consider the user input:

```
soap' UNION SELECT
username,password,NULL FROM
user_table;-- -
```

Being added to the query:

```
query = "SELECT product_name,
product_cost, product_description
FROM product_table WHERE
product_name = " + USER_INPUT
+ "'";
```

Results in:

```
SELECT product_name, product_cost,
product_description FROM
product_table WHERE product_name
= 'soap' UNION SELECT
username,password,NULL FROM
user_table;-- -';
```

This resulting query would expose all the usernames and passwords of the users!

Error-Based Injections

An attacker writes a malicious SQL query to force the application to return an error message with sensitive data. The inside statement of the following SQL query gets the password for the profile ID 1 but throws an error on the value type that should be returned. This error accidentally gives away the password!

```
SELECT user_id FROM users WHERE
username='asdf' UNION select 1, exp(~
(select*from(SELECT Password FROM profiles
WHERE ID=1)x)); -- -
```

Boolean-Based Injections

An attacker takes note of the difference in web responses after sending SQL queries that result in either `TRUE` or `FALSE`. Depending on the result, the HTTP response will change or stay the same. Even though no data is returned from the database, the attacker can gain insight into the database (figure out table names) and eventually build up for a Union-based injection.

```
SELECT username, email FROM users WHERE id
= '1' AND '1' = '1';
```

Time-Based Injections

Makes use of several built-in SQL functions, such as `SLEEP()` and `BENCHMARK()`, to cause visible delays in an application's response time. Like boolean-based injections, this is also used by an attacker to infer information about the database.

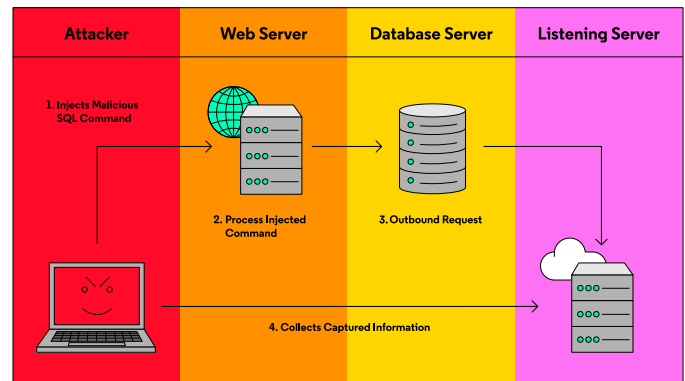
Consider the SQL query in the code block. If there's a 5-second delay before a response from the server, an attacker can confirm the `admin` user's password is

`P@ssw0rd123`.

```
SELECT id FROM users WHERE username = 'a'
OR IF((SELECT password FROM users WHERE
username='admin')='P@ssw0rd123', SLEEP(5),
NULL);-- -';
```

Out-of-Band SQL Injections

Rare and difficult injections to execute for an attacker because the attacker is unable to use the same channel to send the SQL query and gather results. Generally, these SQL injections will cause the database server to send HTTP or DNS requests containing SQL query results to an attacker-controlled server. From there, the attacker could review the log files to identify the query results.



Mitigating SQL Injection Attacks: Prepared Statements

One way SQL injections can be mitigated is through prepared statements. With prepared statements, the query we want to execute is provided to the database in advance. Any input is then treated as a parameter and will not be treated as SQL code.

1. First, a SQL query template is sent to the database. Certain values, called parameters, are left unspecified. For example, user input.
2. The database processes the query and performs optimizations.
3. Values are bound to the parameters and the SQL query is executed.

This method is a nearly foolproof and reliable solution to SQL injections.

```
$username= $_GET['user']; // Set parameter
$stmt = $conn->prepare("SELECT * FROM
Users WHERE name = '?'"); // Prepare
statement
$stmt->bind_param("s", $username); // Bind
parameter to SQL query
$stmt->execute(); // Execute the SQL query
```

Mitigating SQL Injection Attacks: validator.js

The npm package, [validator.js](#), is a library of string validators and sanitizers that includes functions to clean up data inputs, which helps mitigate SQL injections.

Some of the input validators include:

- `isEmail()` - Checks if the input is a valid email address
- `isLength()` - Checks if the input is a certain length
- `isCurrency()` - Checks if the input is currency-formatted
- `isMobilePhone()` - Checks if the input is a valid mobile phone number

Some of the sanitizers include:

- `normalizeEmail()` - Removes formatting on email inputs to remove potentially dangerous characters
- `escape()` - Replaces `<`, `>`, `&`, `'`, and `"` characters that could be confused with HTML entities

```
// Prints "true"
console.log(validator.isCurrency("$42.25"))
```

```
// Prints "student@codecademy.com"
console.log(validator.normalizeEmail("STUDENT@Codecademy.com"))
```

```
// Prints "1 &lt; 2"
console.log(validator.escape("1 < 2"))
```