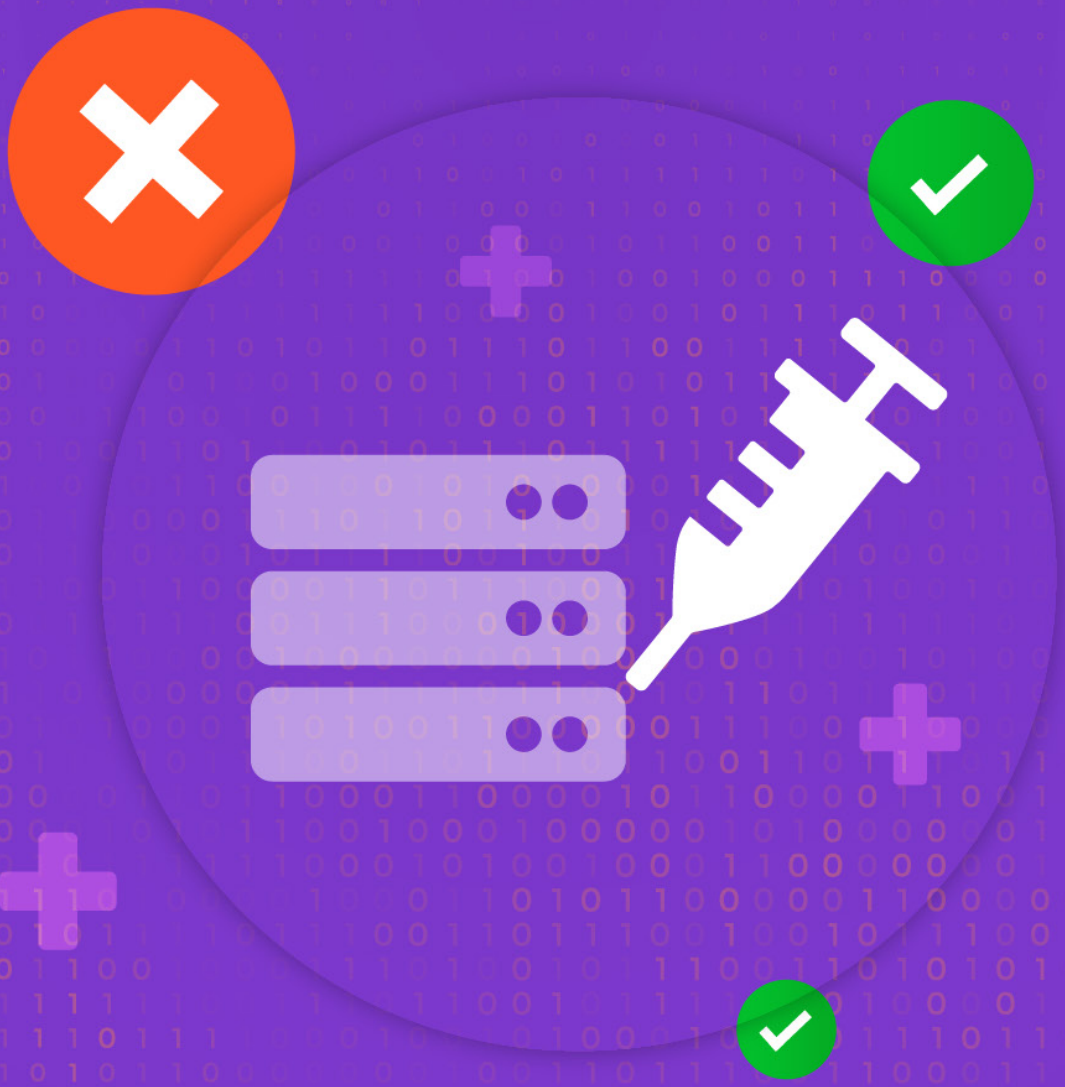


SQL injection cheat sheet: 8 best practices to prevent SQL injection attacks

By: Brian Vermeer

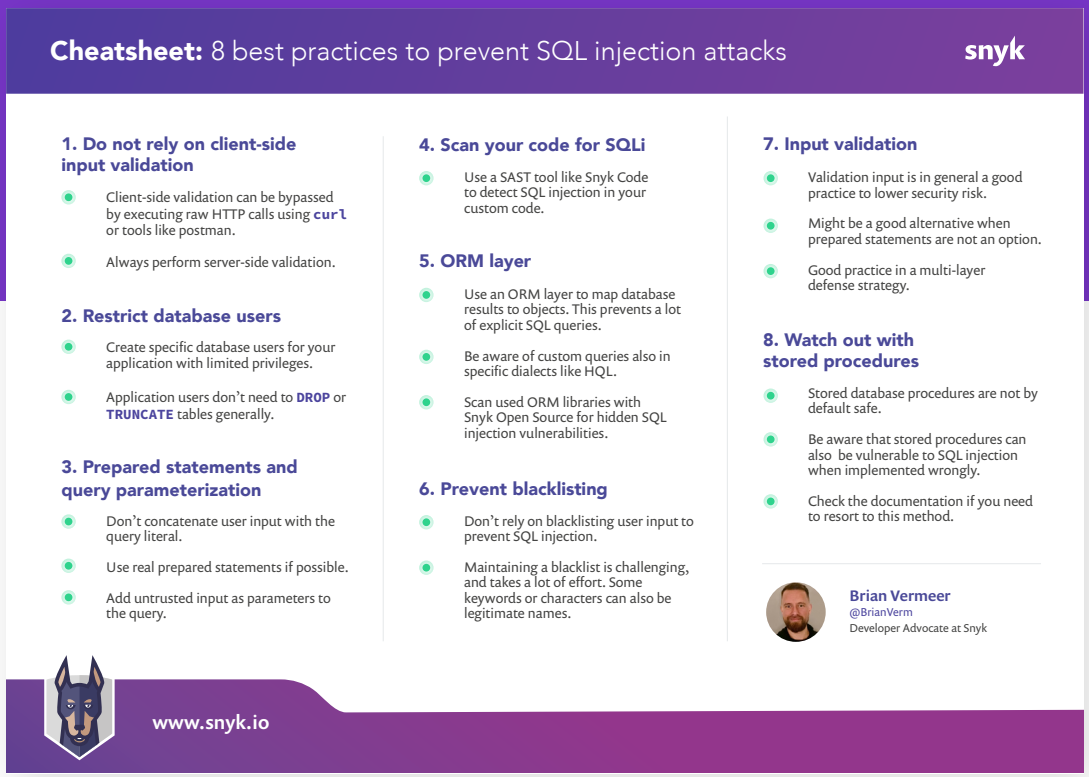
SQL injection is one of the most dangerous vulnerabilities for online applications. It occurs when a user adds untrusted data to a database query. For instance, when filling in a web form. If SQL injection is possible, smart attackers can create user input to steal valuable data, bypass authentication, or corrupt the records in your database.

There are different types of SQL injection attacks, but in general, they all have a similar cause. The untrusted data that the user enters is concatenated with the query string. Therefore the user's input can alter the query's original intent.

Some SQL injection examples are:

- Adding a boolean to a where clause that is always true like `' OR 1=1`
- Escaping part of query by entering line comments `--`
- Ending the initial query and start a new query `'; DROP TABLE USERS;`
- Connecting data from multiple tables by using `UNION`

In this cheatsheet, I will address eight best practices that every application programmer can use to prevent SQL injection attacks. So let's get started to make your application SQLi proof.



1. Do not rely on client-side input validation
2. Use a database user with restricted privileges
3. Use prepared statements and query parameterization
4. Scan your code for SQL injection vulnerabilities
5. Use an ORM layer
6. Don't rely on blocklisting
7. Perform input validation
8. Be careful with stored procedures

1. Do not rely on client-side input validation

Client-side input validation is great. With client-side input validation, you can already prevent that invalid will be sent to your system logic. However, this unfortunately only works for users that do not have bad intentions and want to use the system as designed. To give the user direct feedback that a particular value is not valid is super helpful and user-friendly. Therefore you should be using client-side validation to help your user experience.

When looking at SQL injection, it is not a method you should rely on. You can remove client-side validation by altering some javascript code loaded in your browser. Besides, it is pretty easy to do a basic HTTP call to the backend in a client-server architecture with a parameter that causes a SQL injection. Either by using tools like postman or old-school `curl` commands.

You should validate on the server-side, ideally as close to the source as possible. In this case where you create the SQL query. Everything a client sends you should be considered potentially harmful. So relying on client-side validation for SQL injection, for that matter, is a terrible idea.

2. Use a database user with restricted privileges

There are different types of SQL injection attacks, as mentioned before. Some of them are more harmful than others. Think about it, say my SQL query is something like `"SELECT * FROM USER WHERE USERID = '" + userid +"'"`. The injection `" foo' OR '1'='1 "` will provide all the users and is already harmful. However, `" '; UPDATE message SET password = 'EVIL` " will cause even more problems because the intruder now changed all the entries.

When you create a database user for your application, you have to think about that user's privileges. Does the application need the ability to read, write and update all the databases? How about truncating or dropping tables? If you limit your application's privileges on your database, you can minimize SQL injection's impact. It is probably wise not to have a single database user for your application but make multiple database users and connect them to specific application roles. Security issues are most likely a chain effect, so you should be aware of every link in the chain to prevent heavy damage.

3. Use prepared statements and query parameterization

A lot of languages have built-in features available that help you prevent SQL injection. When writing SQL queries, you can use something like a prepared statement to compile the query. With a prepared statement, we can perform query parameterization. Query parameterization is a technique to create SQL statements dynamically. You create the base query with some placeholders and safely attach the user-given parameters to these placeholders.

When using a real prepared statement and parameterized queries, the database itself actually takes care of the escaping. First, it builds the query execution plan based on the query string
with placeholders. In the second step, the (untrusted) parameters are sent to the database. The query plan is already created, so the parameters do not influence that anymore. This prevents injection altogether.

Java example:

```
String query = "SELECT * FROM USERS WHERE username LIKE ?";
PreparedStatement statement = connection.prepareStatement(query);
statement.setString(1, parameter);
ResultSet result = statement.executeQuery();
```

Python example with MySql connector:

```
cursor = conn.cursor(prepared=True)
params = ("foo",)
cursor.execute("SELECT * FROM USERS WHERE username = %s", params)
```

JavaScript Example with mysql2:

```
connection.query("SELECT * FROM USERS WHERE username = ?",[
     req.body.username
    ],function(error, results){});
//emulates a prepared statement

//OR

connection.execute("SELECT * FROM USERS WHERE username = ?",[
     req.body.username
```

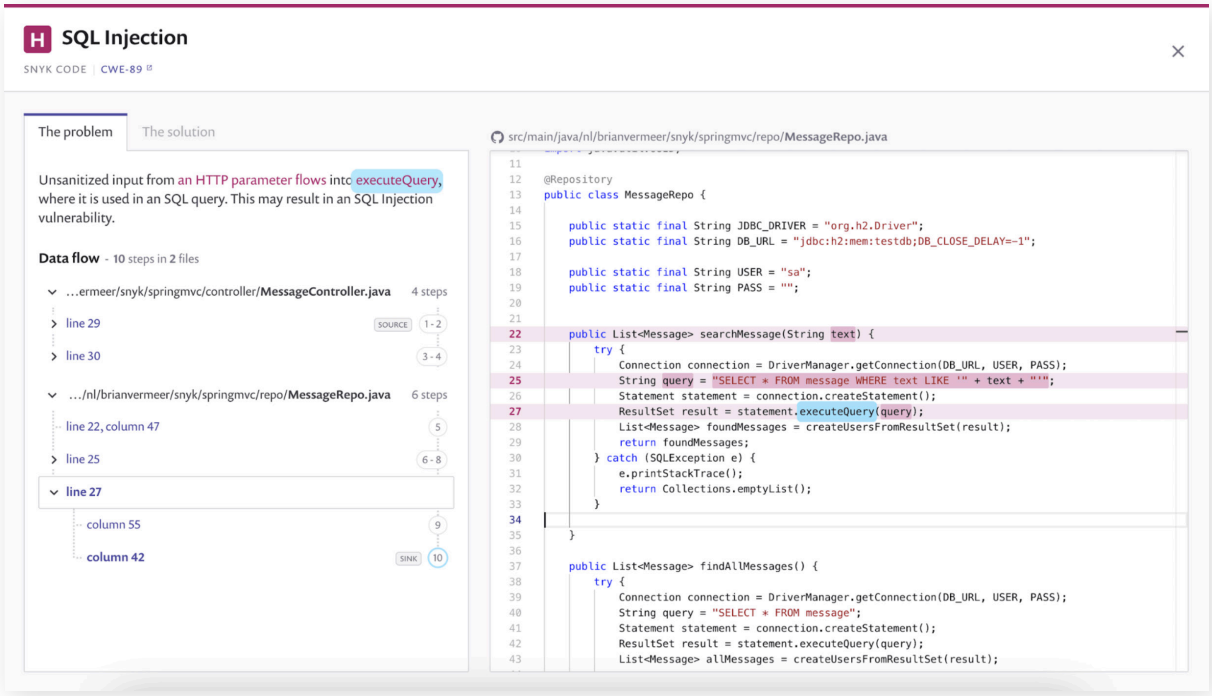
There are multiple ways to do this in JavaScript with, for instance, a MySql database. Be aware when using `.query()`, you do not perform an actual prepared statement. In this case, you parameter substitution is handled on the client-side. So, you are emulating a prepared statement. To make a real prepared statement on the database, you should use the `.execute()` function.

4. Scan your code for SQL injection vulnerabilities

Creating custom code is probably easy. However, mistakes are easily made. To check your code, you might have processes in place like code review and pair programming. However, is the person that reviews your code of pairs with you security savvy. Can that person spot a SQL injection bug in your code? Regardless, it would be nice to automatically examine your custom code for possible security vulnerabilities like SQL injection.

With a SAST (Static Application Security Testing) tool like Snyk Code you can automatically inspect your code for Security vulnerabilities like SQL injection. This can be easily automated in your SDLC by connecting your Git repository to Snyk for instance.

Curious to find out if your code is vulnerable to SQL injection? Scan your code for SQLi vulnerabilities (and get fix advice) for free with Snyk.



5. Use an ORM layer

The use of an object-relational mapping (ORM) layer is also something you can consider. An ORM layer transforms the data from the database into objects and vise-versa. Using an ORM library reduces explicit SQL queries and, therefore, much less vulnerable to SQL injection.

Some great examples of existing ORM libraries are Hibernate for Java and Entity Framework for C#. These languages' strongly typed nature makes it generally possible to generate the mapping between objects and database tables. This way, you don't even need to get involved in any SQL queries yourself.

Nevertheless, the problem here exists if you need to create custom queries. Hibernate for Java has its own Hibernate Query Language (HQL). When compiling HQL queries, you should be aware of injection again and use the `createQuery()` function that works similarly to a prepared statement.

For JavaScript, there are also well-known ORM libraries available like sequelize. With sequelize, you can define how values map to specific types in the database. But let's face it, in the end an ORM library needs to translate the logic back to SQL statements. So, we trust that these libraries implemented proper escaping of the parameters.

To be sure that your ORM library doesn't contain SQL injection problems, you should scan them for known vulnerabilities. Using the wrong, outdated version of sequelize or hibernate will still get you into trouble. Using Snyk Open Source to check your project will prevent you from hidden SQL injection in your libraries and many other problems.

6. Don't rely on blocklisting

I believe this is already familiar to a lot of people, but I will say it once again. Please don't implement blocklists for your parameters. The blocklist approach sets up a collection of rules that define vulnerable input. If the input meets these rules, then the request gets blocked. However, if the ruling is too weak, then a malicious entry will still be effective. If it is too strong, it will block a valid entry.

We, for instance, block every request that contains the word `OR`. This might be a reasonable rule, but it turns out that "Or" is a very common Israeli first name in practice. That means that a bunch of my co-workers will be blocked when inserting their names. The same holds for a single quote `'`. Countless names are containing that character. Think about O'Neill and O'Donnell. But also first names like Dont'a, for example.

7. Perform input validation

Yes, you should do input validation, always! Although prepared statements with query parameterization are the best defense against SQL injection, always create multiple defense layers. Like having limited privileges for a database user, input validation is a great practice to lower your application's risk in general.

Also, there are situations where prepared statements are not available. Some languages don't support this mechanism, or older database systems do not allow you to supply the user input as parameters. Input validation is an acceptable alternative in these cases.

Make sure that input validation relies on allow-listing and not blocklisting, as described earlier. Create a rule that describes all allowed patterns with, for instance, a regular expression, or use a well-maintained library for this. Combine this with prepared statements and query parameterization, and you will have solid defense altogether.

8. Be careful with stored procedures

Many people believe that working with stored procedures is a good way to prevent SQL injection. This is not always the case. Similar to SQL queries created in your application, a stored procedure can also be maliciously injected.

Like SQL queries in your application, you should parameterize the queries in your stored procedure rather than concatenate the parameters. SQL injection in a stored procedure is quite easy to prevent.

So **don't** do this in MySQL:

```
DELIMITER //
CREATE PROCEDURE `FindUsers`(
    IN Username VARCHAR(50)
)
BEGIN

    SET @Statement = CONCAT('SELECT * FROM User WHERE username =
', Username, ' );

    PREPARE stm FROM @Statement;
    EXECUTE stm;
```

Rather use parameterized queries in your stored procedures:

```
DELIMITER //
CREATE PROCEDURE `First`(
    IN Username VARCHAR(50)
)
BEGIN

    PREPARE stm FROM 'SELECT * FROM User WHERE username = ?';
    EXECUTE stm USING Username;

END //
DELIMITER ;
```

As shown above, the same rules apply to stored procedures as to your application's code. The implementation of stored procedures differs between databases. Ensure you know how to implement stored procedures for your database and be mindful about injection there as well. Although I believe that it would be better to have all logic in your application, a stored procedure can be a reasonable solution if prepared statements are not available in the language you develop with.

Cheatsheet: 8 best practices to prevent SQL injection attacks

snyk

1. Do not rely on client-side input validation

- Client-side validation can be bypassed by executing raw HTTP calls using `curl` or tools like postman.
- Always perform server-side validation.

2. Restrict database users

- Create specific database users for your application with limited privileges.
- Application users don't need to `DROP` or `TRUNCATE` tables generally.

3. Prepared statements and query parameterization

- Don't concatenate user input with the query literal.
- Use real prepared statements if possible.
- Add untrusted input as parameters to the query.

4. Scan your code for SQLi

- Use a SAST tool like Snyk Code to detect SQL injection in your custom code.

5. ORM layer

- Use an ORM layer to map database results to objects. This prevents a lot of explicit SQL queries.
- Be aware of custom queries also in specific dialects like HQL.
- Scan used ORM libraries with Snyk Open Source for hidden SQL injection vulnerabilities.

6. Prevent blacklisting

- Don't rely on blacklisting user input to prevent SQL injection.
- Maintaining a blacklist is challenging, and takes a lot of effort. Some keywords or characters can also be legitimate names.

7. Input validation

- Validation input is in general a good practice to lower security risk.
- Might be a good alternative when prepared statements are not an option.
- Good practice in a multi-layer defense strategy.

8. Watch out with stored procedures

- Stored database procedures are not by default safe.
- Be aware that stored procedures can also be vulnerable to SQL injection when implemented wrongly.
- Check the documentation if you need to resort to this method.

Brian Vermeer
@BrianVerm
Developer Advocate at Snyk

www.snyk.io

Ready to secure your app against SQL Injection attacks?

Scan your code for SQL Injection vulnerabilities in seconds. Fix quickly with automated fix advice. Try Snyk for free.