Security Guide

Security Guide

Anonymous · 09/09/2026

Security Guide

`derafu/query` uses prepared statements for all filter values. This page explains the security model, what is and is not protected, and how to use the library safely.

How Values Are Protected

All filter values — the part after the operator symbol in `column?operatorVALUE` — become **named parameters** in the generated SQL:

```
price?>1000
→  price > :param_price_5f3a...
   with parameter binding: :param_price_5f3a... = '1000'
```

The SQL string itself never contains the user-supplied value. It goes through PDO/Doctrine/Illuminate's prepared statement mechanism, which prevents SQL injection in values.

This holds for every operator type: standard comparisons, LIKE patterns, lists, ranges, dates, regex values, and bitwise operands are all bound as parameters.

What SqlSanitizerTrait Protects

`SqlSanitizerTrait` is used for **SQL identifiers** — column names, table names, aliases, and aggregate function names. It strips all characters except `[a-zA-Z0-9_]` from simple identifiers.

Safe uses (identifiers derived from path segments or select expressions):

```
$qb->select('column_name');         // simple identifier
$qb->select('table.column');        // qualified identifier
$qb->select('column AS alias');     // with alias
$qb->select('COUNT(*) AS total');   // aggregate function
```

```
$qb->select('SUM(price) AS total'); // aggregate with column
```

Cases requiring attention:

```
// Complex expressions — arithmetic operators are detected and left as-is.
$qb->select('price * quantity AS total');

// Multi-argument functions — outer function is sanitized; inner args are handled
separately.
$qb->select('COALESCE(column1, column2, 0) AS result');
```

The sanitizer is a defense-in-depth measure for identifier injection, not a substitute for input validation. It strips unexpected characters but cannot reason about business logic (e.g. it does not know which columns a user is allowed to access).

What Is NOT Automatically Protected

Explicit JOIN Conditions

The `join()` / `innerJoin()` / `leftJoin()` / `rightJoin()` methods accept a raw `$condition` string that is inserted verbatim into the SQL:

```
// This condition is NOT sanitized.
$qb->innerJoin('customers', 'i.customer_id = c.id', 'c');
```

Never pass user-supplied input directly as a join condition. Use path-based syntax in `where()` instead — those are parsed and sanitized:

```
// Safe: join condition comes from the path parser.
$qb->where('invoices[alias:i]__customers[on:customer_id=id,alias:c]__name?isnot:null')
;
```

The `?E` Expression Marker

When a filter expression uses `?E` (instead of `?`), the value is treated as a column reference and sanitized as an identifier — **not** bound as a parameter:

```
id?E!=other_alias.id
→  id != other_alias.id   (identifier, no parameter binding)
```

Only use `?E` for trusted, controlled values (e.g. comparing two known column aliases). Never pass user

input as a `?E` value.

`select()` with `$sanitize = false`

```
$qb->select($trustedExpression, sanitize: false);
```

When `$sanitize` is `false`, the expression is inserted without processing. Only use this for pre-validated, application-controlled strings.

Parameters vs. Identifiers

The key distinction in SQL security:

Type	Example	How it's handled
Parameter (value)	`price?>1000` — the `1000`	Bound via prepared statement
Identifier (column/table)	Path segment names, `select()` columns	Sanitized by `SqlSanitizerTrait`

Never try to pass a value as an identifier or vice versa.

Operator Value Validation

Before SQL generation, `FilterParser` validates the raw value against each operator's `pattern` (if defined). Invalid values throw `InvalidArgumentException` immediately:

```
$parser->parse('price?date:not-a-date');   // throws: pattern mismatch
$parser->parse('flags?b&1.5');             // throws: decimal not allowed for binary
op
$parser->parse('status?in:');             // throws: empty list
```

This prevents malformed inputs from reaching the SQL layer, even though the actual injection risk is eliminated by parameter binding.

Input Validation Best Practices

1. **Whitelist columns**: If you accept filter column names from user input (e.g. in an API), validate them against a known-safe list before passing to the expression parser.

2. **Validate operator intent**: Consider whether a user should be allowed to use all operators. For example, regex operators can cause high-CPU queries on large tables; restrict them if necessary.

3. **Limit list sizes**: The `in:` and `notin:` operators accept arbitrarily long lists. Consider validating or capping list length before parsing.

4. **Principle of least privilege**: The database user used by your application should only have `SELECT` (and only `INSERT`/`UPDATE`/`DELETE` where needed). A read-only connection cannot be abused into DDL statements even if SQL injection were somehow possible.

5. **Audit HAVING and JOIN conditions**: `having()` accepts the same expression format as `where()` and is equally safe. Explicit `join()` conditions are not sanitized — see above.

6. **Log and monitor**: Log queries that raise `InvalidArgumentException` — they may indicate probing attempts.

Composite Expression Safety

Composite expressions (`&&`, `||`, `()`) are parsed structurally, not evaluated as SQL. The parser splits on `&&` and `||` at parenthesis depth 0 and recurses — it does not execute or interpolate anything. There is no risk of injection through the composite syntax itself.

```
A?=1&&B?=2      →  two separate bound parameters
(A?=1||B?=2)    →  same, with OR grouping
```

The value inside each leaf expression is always bound as a parameter (unless `?E` is used — see above).

Summary

Area	Protection
Filter values	✅ Prepared statement parameters
Column/table identifiers	✅ `SqlSanitizerTrait` stripping
Operator validation patterns	✅ Regex validation at parse time
Composite expression parsing	✅ Structural parsing, no SQL eval
Explicit `join()` conditions	⚠ Not sanitized — keep application-controlled
`?E` expression references	⚠ Sanitized as identifier — keep application-controlled
`select($expr, sanitize: false)`	⚠ Raw insert — keep application-controlled
Column name whitelisting	❌ Application responsibility
Operator allowlisting	❌ Application responsibility

Last updated on 09/09/2026