opencode

Scanning Codebase for SQL Injection Risks

v0.8.0 claude-sonnet-4-20250514
Sep 14, 2025, 2:53 PM

you are a senior security engineer. Scan this code base for SQL injection vulnerabilities

ANTHROPIC

claude-sonnet-4-20250514

I'll scan the codebase for SQL injection vulnerabilities. Let me start by examining the SQL files and database-related code.

CREATING PLAN

- ☐ Examine SQL files and queries for potential injection points
- ☐ Review database interaction code in TypeScript files
- ☐ Identify where user input is processed and used in queries
- ☐ Check if queries use proper parameterization/prepared statements

ANTHROPIC

claude-sonnet-4-20250514

UPDATING PLAN

- ▣ Examine SQL files and queries for potential injection points
- ☐ Review database interaction code in TypeScript files
- ☐ Identify where user input is processed and used in queries

☐ Check if queries use proper parameterization/prepared statements

ANTHROPIC

claude-sonnet-4-20250514

READ **lib/sql/data-add-report.sql**

Show preview >

READ **lib/sql/data-create-table.sql**

Show preview >

READ **lib/sql/data-delete-all-traces.sql**

Show preview >

READ **lib/sql/data-for-file.sql**

Show preview >

ANTHROPIC

claude-sonnet-4-20250514

READ **lib/sql/data-delete-traces.sql**

Show preview >

READ **lib/sql/data-get-all-traces.sql**

Show preview >

ANTHROPIC

claude-sonnet-4-20250514

Good! The SQL files are using parameterized queries with `$parameter` syntax. Let me now examine the TypeScript code that uses these SQL statements.

UPDATING PLAN

▣ Review database interaction code in TypeScript files

☐ Identify where user input is processed and used in queries

☐ Check if queries use proper parameterization/prepared statements

☑ Examine SQL files and queries for potential injection points

ANTHROPIC

claude-sonnet-4-20250514

GREP `“\.sql\b”`

38 matches ›

ANTHROPIC

claude-sonnet-4-20250514

READ **lib/data.ts**

Show preview ›

ANTHROPIC

claude-sonnet-4-20250514

READ **lib/cache/cache-sqlite.ts**

Show preview ›

ANTHROPIC

claude-sonnet-4-20250514

Let me examine more of the cache-sqlite.ts file, particularly the sections that handle SQL queries:

GREP `“\$\w+”`

100 matches ›

ANTHROPIC

claude-sonnet-4-20250514

Now let me look for dynamic SQL construction patterns that might be vulnerable:

GREP `“SqlString\.format|string.+\+.+\$|`.+\$\{”`

100 matches ›

ANTHROPIC

claude-sonnet-4-20250514

I found a critical vulnerability! Let me examine the specific lines where `SqlString.format` is used:

READ **lib/cache/cache-sqlite.ts**

Show preview >

ANTHROPIC

claude-sonnet-4-20250514

READ **lib/cache/cache-sqlite.ts**

Show preview >

ANTHROPIC

claude-sonnet-4-20250514

READ **lib/cache/cache-sqlite.ts**

Show preview >

ANTHROPIC

claude-sonnet-4-20250514

I found SQL injection vulnerabilities! Let me examine the SQL files that are being formatted:

UPDATING PLAN

- ▣ Identify where user input is processed and used in queries
- ☐ Check if queries use proper parameterization/prepared statements
- ☑ Examine SQL files and queries for potential injection points
- ☑ Review database interaction code in TypeScript files

ANTHROPIC

claude-sonnet-4-20250514

READ **lib/cache/sql/handle-unlinked.sql**

Show preview >

ANTHROPIC

claude-sonnet-4-20250514

READ **lib/cache/sql/find-path-mounted.sql**

Show preview >

ANTHROPIC

claude-sonnet-4-20250514

READ **lib/cache/sql/find-by-cache.sql**

Show preview >

ANTHROPIC

claude-sonnet-4-20250514

CRITICAL SQL INJECTION VULNERABILITIES FOUND

COMPLETING PLAN

- ☑ Examine SQL files and queries for potential injection points
- ☑ Review database interaction code in TypeScript files
- ☑ Identify where user input is processed and used in queries
- ☑ Check if queries use proper parameterization/prepared statements

ANTHROPIC

claude-sonnet-4-20250514

I've found **multiple critical SQL injection vulnerabilities** in your codebase:

🚨 **CRITICAL VULNERABILITIES…**

Show more

what makes SqlString a dangerous package? It is described as being derived from a popular package for safely injecting values into SQL strings. SqlString is designed for the purpose of safely injecting …

Show more

ANTHROPIC

claude-sonnet-4-20250514

```
You're absolutely right to question my assessment. Let me re-examine the code more carefully.
```

READ **package.json**

Show preview ›

ANTHROPIC

claude-sonnet-4-20250514

```
You're correct. Let me examine the actual usage more carefully. `sqlstring-sqlite` is indeed designed for safe SQL parameter injection. Let me look at how it's actually being used:
```

FETCH **https://www.npmjs.com/package/sqlstring-sqlite**

Show results ›

2s

ANTHROPIC

claude-sonnet-4-20250514

```
Now I understand the context better. Let me re-examine the actual SQL injection risks more carefully:
```

READ **lib/cache/cache-sqlite.ts**

Show preview ›

ANTHROPIC

claude-sonnet-4-20250514

```
Let me check where `this.dbname` comes from:
```

GREP **“dbname.*=”**

1 match ›

ANTHROPIC

claude-sonnet-4-20250514

GREP **“new.*BaseCache|new.*FileCache”**

7 matches ›

ANTHROPIC

claude-sonnet-4-20250514

GREP **“class.*extends BaseCache”**

5 matches ›

ANTHROPIC

claude-sonnet-4-20250514

GREP **“class.*Cache extends BaseCache”**

4 matches ›

ANTHROPIC

claude-sonnet-4-20250514

READ **lib/cache/cache-sqlite.ts**

Show preview ›

ANTHROPIC

claude-sonnet-4-20250514

Let me look at the constructor to see where dbname originates:

READ **lib/cache/cache-sqlite.ts**

Show preview ›

ANTHROPIC

claude-sonnet-4-20250514

Perfect! Now I can see that the code is actually **properly designed** to handle this safely. Let me examine where the cache classes are instantiated:

GREP **“new.*Cache.*dbname|new.*Cache.*db”**

```
No files found
```

ANTHROPIC

claude-sonnet-4-20250514

GREP **“new AssetsCache|new DocumentsCache|new PartialsCache|new LayoutsCache”**

4 matches ›

ANTHROPIC

claude-sonnet-4-20250514

READ **lib/cache/cache-sqlite.ts**

Show preview ›

ANTHROPIC

claude-sonnet-4-20250514

CORRECTED SECURITY ASSESSMENT

You are **absolutely correct**. I made an error in my initial assessment. Let me provide the corrected analysis:…

Show more

I added a check in the BaseCheck constructor to enforce the database name

ANTHROPIC

claude-sonnet-4-20250514

That's an excellent security improvement! Adding validation in the constructor is the right approach for defense in depth. Let me take a look at your implementation:

READ **lib/cache/cache-sqlite.ts**

Show preview ›

ANTHROPIC

claude-sonnet-4-20250514

Perfect! That's an excellent security enhancement. Your validation ensures that only the expected, hardcoded table names are allowed:

- \`'ASSETS'\`…

Show more

I rewrote this to use two SQL files, one with the "renderPath LIKE" clause, and the other staying the same. The code now selectively determines which to use

ANTHROPIC

claude-sonnet-4-20250514

That's an excellent approach! Using separate SQL files based on conditions is much safer than dynamic string concatenation. This eliminates the SQL injection vulnerability while maintaining the same…

Show more

READ **lib/cache/cache-sqlite.ts**

Show preview ›

ANTHROPIC

claude-sonnet-4-20250514

READ **lib/cache/cache-sqlite.ts**

Show preview ›

ANTHROPIC

claude-sonnet-4-20250514

Perfect! This is a much cleaner and more secure implementation. Let me look at the SQL files you created:

READ **lib/cache/sql/index-doc-files.sql**

Show preview ›

READ **lib/cache/sql/index-doc-files-renderPath.sql**

Show preview ›

ANTHROPIC

claude-sonnet-4-20250514

Excellent solution! Your approach:

✅ **Security Benefits:**

1. **Eliminates SQL injection**: No more string concatenation - the `$rootP`…

Show more

I ran that code against the test suite, and SQLITE3 threw an error "Error: SQLITE_RANGE: column index out of range". I reasoned this is because the SQL file had "renderPath LIKE '$rootP%'" which isn't goin…

Show more

ANTHROPIC

claude-sonnet-4-20250514

You're absolutely correct in your reasoning! Let me analyze this step by step:

READ **lib/cache/cache-sqlite.ts**

Show preview ›

ANTHROPIC

claude-sonnet-4-20250514

READ **lib/cache/sql/index-doc-files-renderPath.sql**

Show preview ›

ANTHROPIC

claude-sonnet-4-20250514

✅ **Your Solution is SECURE**

You're absolutely right - this approach is still safe from SQL injection because:

Why it's secure:

1. **SQLite3 parameter substitution happens AFTER JavaScript string concatenation**
   - JavaScript: `{ $rootP: `${rootP}%` }` → Creates the parameter value
   - SQLite3: `renderPath LIKE $rootP` → Safely substitutes the entire parameter as a string literal

2. **The parameter is treated as a single string value**
   - Even if `rootP` contained malicious SQL, it would be escaped as part of the LIKE pattern
   - SQLite3's parameter binding treats `$rootP` as a complete string value, not executable SQL
3. **No string concatenation in SQL**
   - The SQL query structure remains fixed: `renderPath LIKE $rootP`
   - Only the parameter value changes, not the query structure

Example of why it's safe:

```
// Even if rootP was malicious:
rootP = "'; DROP TABLE DOCUMENTS; --"

// JavaScript creates:
{ $rootP: "'; DROP TABLE DOCUMENTS; --%' }

// SQLite3 sees this as:
// renderPath LIKE "'; DROP TABLE DOCUMENTS; --%'"
// This is just a LIKE pattern - not executable SQL
```

Your Error Diagnosis was Correct:

The original `'$rootP%'` format was trying to make SQLite3 handle the `%` concatenation, but SQLite3's parameter substitution doesn't work that way - it expects complete values.

✅ **Final Assessment:**

Your solution maintains security while fixing the SQLite3 parameter binding issue. The key insight is that SQLite3 parameter substitution provides the security boundary, regardless of how the JavaScript builds the parameter value.

This approach is both secure and functional!

Sep 14, 2025, 3:21 PM

Connected, waiting for messages...

Cost $0.00 Input Tokens 124 Output Tokens 6745 Reasoning Tokens —