Contents

1 SEC-003: SQL Injection

1.1 Severity

Critical

1.2 OWASP Reference

A03:2021 - Injection

1.3 CWE References

- CWE-89: SQL Injection

1.4 Description

SQL Injection vulnerabilities occur when an application constructs SQL queries using untrusted input without proper sanitization or parameterization. In MCP servers that interact with databases, this allows attackers to manipulate SQL queries to read, modify, or delete data, bypass authentication, execute administrative operations, or in some cases achieve remote code execution on the database server. This represents one of the most prevalent and dangerous vulnerabilities in web applications and API servers.

1.5 Compliance Mapping

1.5.1 SOC2

- **CC6.1**: Logical and Physical Access Controls - The organization implements access controls to protect against unauthorized database access.
- **CC6.7**: Data Classification - The organization restricts access to sensitive data based on classification levels.
- **CC7.2**: System Monitoring - The organization monitors database access patterns and detects anomalous queries.

1.5.2 HIPAA

- **§164.312(a)(1)**: Access Control - Implement technical policies to restrict access to ePHI in databases.
- **§164.312(b)**: Audit Controls - Implement mechanisms to record and examine access to systems containing ePHI.
- **§164.312(c)(1)**: Integrity Controls - Protect ePHI from improper alteration through secure database operations.
- **§164.312(e)(1)**: Transmission Security - Implement technical security measures to guard against unauthorized database access.

1.5.3 PCI DSS

- **6.2.4**: All bespoke and custom software is developed securely
- **6.5.1**: Injection flaws, particularly SQL injection, are addressed through secure coding
- **8.2.1**: Strong cryptography is used to render authentication credentials unreadable during transmission and storage
- **10.2.4**: All invalid logical access attempts are logged
- **11.3.1.2**: SQL injection vulnerabilities are identified through automated scanning

1.6 Vulnerable Code Example

```
// □ INSECURE: Direct string concatenation in SQL queries
const express = require('express');
const mysql = require('mysql2');
const app = express();

app.use(express.json());

const db = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: 'password',
  database: 'mcp_data'
});

// Vulnerable endpoint: user authentication
app.post('/api/login', (req, res) => {
  const { username, password } = req.body;

  // DANGER: Direct string concatenation allows SQL injection
  // Attacker input: username = "admin' OR '1'='1' --"
  const query = `SELECT * FROM users WHERE username = '${username}' AND password =

  db.query(query, (err, results) => {
    if (err) {
      return res.status(500).json({ error: 'Database error' });
```

```
    }
    if (results.length > 0) {
      res.json({ success: true, user: results[0] });
    } else {
      res.status(401).json({ error: 'Invalid credentials' });
    }
  });
});

// Vulnerable endpoint: data retrieval
app.get('/api/users/:id', (req, res) => {
  const userId = req.params.id;

  // DANGER: No input validation or parameterization
  // Attacker input: id = "1 UNION SELECT credit_card, ssn, password FROM sensitive
  const query = `SELECT id, username, email FROM users WHERE id = ${userId}`;

  db.query(query, (err, results) => {
    if (err) {
      return res.status(500).json({ error: err.message });
    }
    res.json(results);
  });
});

// Vulnerable endpoint: search functionality
app.get('/api/search', (req, res) => {
  const { term } = req.query;

  // DANGER: Using LIKE with unescaped input
  // Attacker input: term = "%' OR 1=1; DROP TABLE users; --"
  const query = `SELECT * FROM documents WHERE title LIKE '%${term}%'`;

  db.query(query, (err, results) => {
    if (err) {
      return res.status(500).json({ error: 'Search failed' });
    }
    res.json(results);
  });
});

app.listen(3000);
```

1.7 Secure Implementation

```
// □ SECURE: Parameterized queries with input validation
const express = require('express');
```

```
const mysql = require('mysql2/promise');
const bcrypt = require('bcrypt');
const app = express();

app.use(express.json());

// Use connection pool for better performance and security
const pool = mysql.createPool({
  host: 'localhost',
  user: 'mcp_app_user', // Use limited privilege account
  password: process.env.DB_PASSWORD,
  database: 'mcp_data',
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0
});

// Helper function to validate numeric input
function validateNumericId(id) {
  const numId = parseInt(id, 10);
  if (isNaN(numId) || numId < 1) {
    throw new Error('Invalid ID format');
  }
  return numId;
}

// Secure endpoint: user authentication with parameterized query
app.post('/api/login', async (req, res) => {
  const { username, password } = req.body;

  // Input validation
  if (!username || !password || username.length > 50) {
    return res.status(400).json({ error: 'Invalid input' });
  }

  try {
    // Use parameterized query with placeholders (?)
    // The database driver handles escaping automatically
    const [rows] = await pool.execute(
      'SELECT id, username, password_hash, role FROM users WHERE username = ? LIMI
      [username]
    );

    if (rows.length === 0) {
      return res.status(401).json({ error: 'Invalid credentials' });
    }

    const user = rows[0];
```

```
    // Verify password using bcrypt (never store plain text passwords)
    const isValidPassword = await bcrypt.compare(password, user.password_hash);

    if (!isValidPassword) {
      return res.status(401).json({ error: 'Invalid credentials' });
    }

    // Don't return password hash
    delete user.password_hash;
    res.json({ success: true, user });

  } catch (error) {
    console.error('Login error:', error);
    res.status(500).json({ error: 'Authentication failed' });
  }
});

// Secure endpoint: data retrieval with validation
app.get('/api/users/:id', async (req, res) => {
  try {
    // Validate and sanitize input
    const userId = validateNumericId(req.params.id);

    // Use parameterized query
    const [rows] = await pool.execute(
      'SELECT id, username, email, created_at FROM users WHERE id = ?',
      [userId]
    );

    if (rows.length === 0) {
      return res.status(404).json({ error: 'User not found' });
    }

    res.json(rows[0]);

  } catch (error) {
    if (error.message === 'Invalid ID format') {
      return res.status(400).json({ error: error.message });
    }
    console.error('Database error:', error);
    res.status(500).json({ error: 'Failed to retrieve user' });
  }
});

// Secure endpoint: search with parameterized query
app.get('/api/search', async (req, res) => {
  const { term } = req.query;
```

```
  // Input validation
  if (!term || typeof term !== 'string' || term.length > 100) {
    return res.status(400).json({ error: 'Invalid search term' });
  }

  try {
    // Use parameterized query for LIKE operations
    // The % wildcards are added in the parameter, not concatenated
    const searchTerm = `%${term}%`;

    const [rows] = await pool.execute(
      'SELECT id, title, summary, created_at FROM documents WHERE title LIKE ? LIM
      [searchTerm]
    );

    res.json({ results: rows, count: rows.length });

  } catch (error) {
    console.error('Search error:', error);
    res.status(500).json({ error: 'Search failed' });
  }
});

// Secure endpoint: bulk operations with transactions
app.post('/api/users/bulk-update', async (req, res) => {
  const { updates } = req.body; // Array of {id, email} objects

  if (!Array.isArray(updates) || updates.length === 0 || updates.length > 100) {
    return res.status(400).json({ error: 'Invalid updates array' });
  }

  const connection = await pool.getConnection();

  try {
    await connection.beginTransaction();

    for (const update of updates) {
      const userId = validateNumericId(update.id);

      // Validate email format
      if (!/^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(update.email)) {
        throw new Error('Invalid email format');
      }

      // Parameterized update query
      await connection.execute(
        'UPDATE users SET email = ?, updated_at = NOW() WHERE id = ?',
```

```
        [update.email, userId]
      );
    }

    await connection.commit();
    res.json({ success: true, updated: updates.length });

  } catch (error) {
    await connection.rollback();
    console.error('Bulk update error:', error);
    res.status(500).json({ error: 'Bulk update failed' });
  } finally {
    connection.release();
  }
});

app.listen(3000);
```

1.8 Remediation Steps

1. **Always Use Parameterized Queries (Prepared Statements)**: Never concatenate user input directly into SQL queries. Use parameterized queries with placeholders (? or named parameters) provided by your database library. This ensures the database treats user input as data, not executable SQL code, completely preventing SQL injection.
2. **Implement Input Validation and Type Checking**: Validate all user input before using it in queries. For numeric IDs, parse and validate they are integers. For strings, check length limits and format. Use whitelists for allowed values when possible. Reject any input that doesn't match expected patterns.
3. **Apply Principle of Least Privilege to Database Accounts**: Create separate database accounts for your application with minimal required permissions. Never use the root or admin account. Grant only SELECT, INSERT, UPDATE, DELETE on specific tables needed. Revoke permissions for DROP, CREATE, ALTER, and administrative functions.
4. **Implement Multiple Layers of Defense**: Use Web Application Firewalls (WAF) with SQL injection detection rules. Enable database query logging and monitoring for suspicious patterns. Use stored procedures where appropriate (but still parameterize inputs to stored procedures). Implement rate limiting on database-intensive endpoints. Conduct regular security testing and code reviews.

1.9 References

- OWASP SQL Injection
- OWASP SQL Injection Prevention Cheat Sheet
- CWE-89: SQL Injection
- Node.js mysql2 Prepared Statements

- NIST SP 800-53 SI-10: Information Input Validation
- PortSwigger SQL Injection Cheat Sheet

Last Updated: January 2026
Status: Published
Language: English