Contents

1 SEC-003: Inyección SQL

1.1 Severidad

Crítica

1.2 Referencia OWASP

A03:2021 - Inyección

1.3 Referencias CWE

- CWE-89: Inyección SQL

1.4 Descripción

Las vulnerabilidades de inyección SQL ocurren cuando una aplicación construye consultas SQL usando entradas no confiables sin la sanitización o parametrización adecuada. En servidores MCP que interactúan con bases de datos, esto permite a los atacantes manipular consultas SQL para leer, modificar o eliminar datos, evitar la autenticación, ejecutar operaciones administrativas o, en algunos casos, lograr ejecución remota de código en el servidor de base de datos. Esta representa una de las vulnerabilidades más prevalentes y peligrosas en aplicaciones web y servidores API.

1.5 Mapeo de Cumplimiento

1.5.1 SOC2

- **CC6.1**: Controles de Acceso Lógico y Físico - La organización implementa controles de acceso para proteger contra acceso no autorizado a bases de datos.
- **CC6.7**: Clasificación de Datos - La organización restringe el acceso a datos sensibles basándose en niveles de clasificación.
- **CC7.2**: Monitoreo del Sistema - La organización monitorea patrones de acceso a bases de datos y detecta consultas anómalas.

1.5.2 HIPAA

- **§164.312(a)(1)**: Control de Acceso - Implementar políticas técnicas para restringir el acceso a ePHI en bases de datos.
- **§164.312(b)**: Controles de Auditoría - Implementar mecanismos para registrar y examinar el acceso a sistemas que contienen ePHI.
- **§164.312(c)(1)**: Controles de Integridad - Proteger ePHI de alteración indebida a través de operaciones seguras de base de datos.
- **§164.312(e)(1)**: Seguridad de Transmisión - Implementar medidas de seguridad técnica para proteger contra acceso no autorizado a bases de datos.

1.5.3 PCI DSS

- **6.2.4**: Todo el software personalizado y a medida se desarrolla de forma segura
- **6.5.1**: Las fallas de inyección, particularmente inyección SQL, se abordan mediante codificación segura
- **8.2.1**: Se utiliza criptografía fuerte para hacer ilegibles las credenciales de autenticación durante la transmisión y almacenamiento
- **10.2.4**: Todos los intentos de acceso lógico inválidos se registran
- **11.3.1.2**: Las vulnerabilidades de inyección SQL se identifican mediante escaneo automatizado

1.6 Ejemplo de Código Vulnerable

```
// □ INSEGURO: Concatenación directa de cadenas en consultas SQL
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'
});

// Endpoint vulnerable: autenticación de usuario
app.post('/api/login', (req, res) => {
  const { username, password } = req.body;

  // PELIGRO: La concatenación directa de cadenas permite inyección SQL
  // Entrada del atacante: 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: 'Error de base de datos' });
    }
    if (results.length > 0) {
      res.json({ success: true, user: results[0] });
    } else {
      res.status(401).json({ error: 'Credenciales inválidas' });
    }
  });
});

// Endpoint vulnerable: recuperación de datos
app.get('/api/users/:id', (req, res) => {
  const userId = req.params.id;

  // PELIGRO: Sin validación de entrada ni parametrización
  // Entrada del atacante: id = "1 UNION SELECT credit_card, ssn, password FROM se
  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);
  });
});

// Endpoint vulnerable: funcionalidad de búsqueda
app.get('/api/search', (req, res) => {
  const { term } = req.query;

  // PELIGRO: Usando LIKE con entrada sin escapar
  // Entrada del atacante: 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: 'Búsqueda fallida' });
    }
    res.json(results);
  });
});

app.listen(3000);
```

1.7 Implementación Segura

```
// ✅ SEGURO: Consultas parametrizadas con validación de entrada
const express = require('express');
const mysql = require('mysql2/promise');
const bcrypt = require('bcrypt');
const app = express();

app.use(express.json());

// Usar pool de conexiones para mejor rendimiento y seguridad
const pool = mysql.createPool({
  host: 'localhost',
  user: 'mcp_app_user', // Usar cuenta con privilegios limitados
  password: process.env.DB_PASSWORD,
  database: 'mcp_data',
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0
});

// Función auxiliar para validar entrada numérica
function validateNumericId(id) {
  const numId = parseInt(id, 10);
  if (isNaN(numId) || numId < 1) {
    throw new Error('Formato de ID inválido');
  }
  return numId;
}

// Endpoint seguro: autenticación de usuario con consulta parametrizada
app.post('/api/login', async (req, res) => {
  const { username, password } = req.body;

  // Validación de entrada
  if (!username || !password || username.length > 50) {
    return res.status(400).json({ error: 'Entrada inválida' });
  }

  try {
    // Usar consulta parametrizada con placeholders (?)
    // El driver de base de datos maneja el escapado automáticamente
    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: 'Credenciales inválidas' });
    }

    const user = rows[0];

    // Verificar contraseña usando bcrypt (nunca almacenar contraseñas en texto pl
    const isValidPassword = await bcrypt.compare(password, user.password_hash);

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

    // No devolver el hash de contraseña
    delete user.password_hash;
    res.json({ success: true, user });

  } catch (error) {
    console.error('Error de login:', error);
    res.status(500).json({ error: 'Autenticación fallida' });
  }
});

// Endpoint seguro: recuperación de datos con validación
app.get('/api/users/:id', async (req, res) => {
  try {
    // Validar y sanitizar entrada
    const userId = validateNumericId(req.params.id);

    // Usar consulta parametrizada
    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: 'Usuario no encontrado' });
    }

    res.json(rows[0]);

  } catch (error) {
    if (error.message === 'Formato de ID inválido') {
      return res.status(400).json({ error: error.message });
    }
    console.error('Error de base de datos:', error);
    res.status(500).json({ error: 'Fallo al recuperar usuario' });
  }
});
```

```
// Endpoint seguro: búsqueda con consulta parametrizada
app.get('/api/search', async (req, res) => {
  const { term } = req.query;

  // Validación de entrada
  if (!term || typeof term !== 'string' || term.length > 100) {
    return res.status(400).json({ error: 'Término de búsqueda inválido' });
  }

  try {
    // Usar consulta parametrizada para operaciones LIKE
    // Los comodines % se agregan en el parámetro, no concatenados
    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('Error de búsqueda:', error);
    res.status(500).json({ error: 'Búsqueda fallida' });
  }
});

// Endpoint seguro: operaciones en lote con transacciones
app.post('/api/users/bulk-update', async (req, res) => {
  const { updates } = req.body; // Array de objetos {id, email}

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

  const connection = await pool.getConnection();

  try {
    await connection.beginTransaction();

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

      // Validar formato de email
      if (!/^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(update.email)) {
        throw new Error('Formato de email inválido');
      }
```

```
      // Consulta de actualización parametrizada
      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('Error de actualización en lote:', error);
    res.status(500).json({ error: 'Actualización en lote fallida' });
  } finally {
    connection.release();
  }
});

app.listen(3000);
```

1.8 Pasos de Remediación

1. **Siempre Usar Consultas Parametrizadas (Prepared Statements)**: Nunca concatenar entradas de usuario directamente en consultas SQL. Usar consultas parametrizadas con placeholders (? o parámetros nombrados) proporcionados por tu librería de base de datos. Esto asegura que la base de datos trate la entrada del usuario como datos, no como código SQL ejecutable, previniendo completamente la inyección SQL.

2. **Implementar Validación de Entrada y Verificación de Tipos**: Validar toda entrada de usuario antes de usarla en consultas. Para IDs numéricos, parsear y validar que son enteros. Para cadenas, verificar límites de longitud y formato. Usar listas blancas para valores permitidos cuando sea posible. Rechazar cualquier entrada que no coincida con los patrones esperados.

3. **Aplicar Principio de Mínimo Privilegio a Cuentas de Base de Datos**: Crear cuentas de base de datos separadas para tu aplicación con los permisos mínimos requeridos. Nunca usar la cuenta root o admin. Otorgar solo SELECT, INSERT, UPDATE, DELETE en tablas específicas necesarias. Revocar permisos para DROP, CREATE, ALTER y funciones administrativas.

4. **Implementar Múltiples Capas de Defensa**: Usar Web Application Firewalls (WAF) con reglas de detección de inyección SQL. Habilitar logging y monitoreo de consultas de base de datos para patrones sospechosos. Usar procedimientos almacenados cuando sea apropiado (pero aún parametrizar entradas a procedimientos almacenados). Implementar limitación de tasa en endpoints intensivos en base de datos. Realizar pruebas de seguridad y revisiones de código regulares.

1.9 Referencias

- OWASP SQL Injection
- OWASP SQL Injection Prevention Cheat Sheet
- CWE-89: Inyección SQL
- Node.js mysql2 Prepared Statements
- NIST SP 800-53 SI-10: Validación de Entrada de Información
- PortSwigger SQL Injection Cheat Sheet

Última Actualización: Enero 2026
Estado: Publicado
Idioma: Español