SQL Injection y Seguridad

Bases de Datos 1 (CC3088) - 2026

SQL Injection y Seguridad

Semestre 01, 2026

Problemática

Una aplicación web recibe el nombre de usuario y contraseña.

Construye una consulta SQL con esos datos y la ejecuta.

```
SELECT * FROM usuarios
WHERE username = 'admin' AND password = '1234';
```

Funciona perfectamente en condiciones normales.

¿Qué pasa si el usuario escribe algo inesperado?

El input del atacante

El atacante escribe como nombre de usuario:

```
admin' --
```

La consulta resultante:

```
SELECT * FROM usuarios
WHERE username = 'admin' --' AND password = 'lo que sea';
```

`--` comienza un comentario en SQL.

Todo lo que viene después es ignorado.

La condición de contraseña desaparece.

El atacante entra al sistema sin conocer la contraseña.

La causa raíz

La aplicación mezcla datos del usuario con código SQL.

El motor de base de datos no distingue entre ambos.

Lo que era un dato se convierte en parte de la consulta.

SQL Injection explota exactamente esa confusión.

Definición

SQL Injection es una vulnerabilidad que permite a un atacante insertar o "inyectar" código SQL dentro de una consulta legítima.

El atacante controla parte del SQL que se ejecuta en la base de datos.

Por qué es tan grave

- Es una de las vulnerabilidades más antiguas y más explotadas de la historia.
- Aparece consistentemente en el OWASP Top 10, la lista de las vulnerabilidades web más críticas.

- Puede comprometer la confidencialidad, integridad y disponibilidad de los datos.
- Una sola vulnerabilidad puede exponer toda la base de datos.

El vector de ataque

Cualquier punto donde la aplicación incorpora datos del usuario en una consulta SQL es un vector potencial:

- Formularios de login
- Campos de búsqueda
- Parámetros en la URL
- Cabeceras HTTP
- Cookies

Ataque 1: Bypass de autenticación

Código vulnerable (Python)

```
username = input("Usuario: ")
password = input("Contraseña: ")

query = "SELECT * FROM usuarios WHERE username = '" + username + "' AND password = '" + p

resultado = db.execute(query)

if resultado:
    print("Acceso concedido")
```

Input malicioso

```
Usuario:    admin' --
Contraseña: (cualquier cosa)
```

Consulta resultante

```
SELECT * FROM usuarios
WHERE username = 'admin' --' AND password = 'x';
```

El motor ejecuta:

```
SELECT * FROM usuarios
WHERE username = 'admin'
```

Si existe un usuario `admin`, el atacante obtiene acceso.

Variantes del bypass

```
' OR '1'='1
' OR 1=1 --
' OR 'x'='x
admin'/*
```

Todas logran lo mismo: hacer que la condición WHERE sea siempre verdadera.

```
SELECT * FROM usuarios
WHERE username = '' OR '1'='1' AND password = '';
-- Devuelve todos los usuarios
```

Ataque 2: Extracción de datos

El atacante no solo quiere entrar. Quiere los datos.

UNION-based injection

Si la aplicación muestra resultados de la consulta en pantalla:

```
URL: /productos?categoria=electronica' UNION SELECT null,username,password FROM usuarios
```

La consulta original:

```
SELECT id, nombre, precio FROM productos
WHERE categoria = 'electronica'
```

La consulta inyectada:

```
SELECT id, nombre, precio FROM productos
WHERE categoria = 'electronica'
UNION SELECT null, username, password FROM usuarios --
```

La aplicación muestra los productos **y** los usuarios con sus contraseñas.

Alcance

- Nombres de todas las tablas: `information_schema.tables`
- Columnas de cualquier tabla: `information_schema.columns`
- Cualquier dato almacenado en la base de datos
- En algunos casos: archivos del servidor

Ataque 3: Modificación y destrucción

Inyección en UPDATE

```
nombre = input("Nuevo nombre: ")
query = "UPDATE usuarios SET nombre = '" + nombre + "' WHERE id = 5"
```

Input malicioso:

```
admin', rol = 'superadmin' WHERE id = 5 --
```

Consulta resultante:

```
UPDATE usuarios SET nombre = 'admin', rol = 'superadmin' WHERE id = 5 --' WHERE id = 5
```

El atacante se asigna a sí mismo privilegios de administrador.

El peor caso

```
'; DROP TABLE usuarios; --
```

```
SELECT * FROM usuarios WHERE username = ''; DROP TABLE usuarios; --'
```

Dependiendo de la configuración, esto puede ejecutar múltiples sentencias.

La tabla de usuarios es eliminada.

Tipos de SQL Injection

In-band (clásica)

El resultado de la inyección se muestra directamente en la respuesta de la aplicación.

Es el tipo más fácil de explotar porque el atacante ve inmediatamente el resultado.

Incluye error-based (usa mensajes de error) y UNION-based.

Blind (ciega)

La aplicación no muestra el resultado de la consulta ni errores.

El atacante hace preguntas de sí/no y deduce la información por el comportamiento de la aplicación.

```
¿El primer carácter del password es 'a'? → la página carga normal = sí
¿El primer carácter del password es 'b'? → la página da error = no
```

Lento pero igualmente efectivo. Automatizable.

Time-based blind

Cuando ni siquiera hay diferencia observable en el comportamiento, el atacante usa el tiempo.

```
'; SELECT CASE WHEN (1=1) THEN pg_sleep(5) ELSE pg_sleep(0) END --
```

Si la página tarda 5 segundos en responder: la condición es verdadera.

El atacante extrae datos bit a bit midiendo tiempos de respuesta.

Out-of-band

Usa canales externos (DNS, HTTP) para exfiltrar datos.

Menos común, requiere configuración específica del servidor.

La raíz del problema

SQL Injection no es un problema de validación de datos.

Es un problema de **separación de código y datos**.

El motor no sabe la diferencia

Para PostgreSQL, esta consulta:

```
SELECT * FROM usuarios WHERE username = 'admin' --'
```

es sintácticamente válida.

El motor no sabe que `--` fue inyectado por un atacante.

Ejecuta lo que recibe.

El error de diseño

Construir consultas concatenando strings mezcla el código (la consulta) con los datos (el input).

Una vez mezclados, es imposible distinguirlos.

La solución no es "limpiar mejor el input".

La solución es **nunca mezclarlos**.

Prevención: Consultas parametrizadas

La defensa principal y más efectiva.

Se separa la estructura de la consulta de los datos del usuario.

El motor recibe la consulta y los datos por separado. Nunca los mezcla.

Código vulnerable vs. seguro (Python con psycopg2)

Vulnerable:

```
query = "SELECT * FROM usuarios WHERE username = '" + username + "'"
cursor.execute(query)
```

Seguro:

```
query = "SELECT * FROM usuarios WHERE username = %s"
cursor.execute(query, (username,))
```

Cómo funciona

Con consultas parametrizadas, el motor compila la consulta primero.

Luego inserta los datos en los espacios reservados.

Los datos nunca son interpretados como código SQL.

```
Input del atacante: admin' --

Motor recibe:
  Consulta: SELECT * FROM usuarios WHERE username = $1
  Dato:     "admin' --"

Resultado: busca el usuario cuyo nombre es literalmente "admin' --"
```

El apóstrofo y el guion doble son tratados como texto, no como SQL.

Consultas parametrizadas en diferentes contextos

PostgreSQL nativo con psycopg2:

```
cursor.execute(
    "SELECT * FROM productos WHERE categoria = %s AND precio < %s",
    (categoria, precio_max)
```

```
)
```

Con múltiples parámetros:

```
cursor.execute(
    "INSERT INTO pedidos (cliente_id, producto_id, cantidad) VALUES (%s, %s, %s)",
    (cliente_id, producto_id, cantidad)
)
```

Nunca usar f-strings ni concatenación para construir consultas:

```
# MAL — siempre vulnerable
query = f"SELECT * FROM usuarios WHERE id = {user_id}"

# BIEN — siempre seguro
query = "SELECT * FROM usuarios WHERE id = %s"
cursor.execute(query, (user_id,))
```

Prevención: Funciones almacenadas

Las funciones en PostgreSQL separan la lógica SQL de la aplicación.

La aplicación solo llama a la función con parámetros. No construye SQL.

Función para autenticación

```
CREATE OR REPLACE FUNCTION autenticar_usuario(
    p_username TEXT,
    p_password TEXT
)
RETURNS BOOLEAN AS $$
DECLARE
    v_hash TEXT;
BEGIN
```

```
    SELECT password_hash INTO v_hash
    FROM usuarios
    WHERE username = p_username;

    IF NOT FOUND THEN
        RETURN FALSE;
    END IF;

    RETURN v_hash = crypt(p_password, v_hash);
END;
$$ LANGUAGE plpgsql;
```

La aplicación llama:

```
cursor.execute("SELECT autenticar_usuario(%s, %s)", (username, password))
resultado = cursor.fetchone()[0]
```

El SQL real nunca lo construye la aplicación.

Función para búsqueda de productos

```
CREATE OR REPLACE FUNCTION buscar_productos(
    p_categoria TEXT,
    p_precio_max NUMERIC
)
RETURNS TABLE(id INT, nombre TEXT, precio NUMERIC) AS $$
BEGIN
    RETURN QUERY
    SELECT p.id, p.nombre, p.precio
    FROM productos p
    WHERE p.categoria = p_categoria
      AND p.precio <= p_precio_max;
END;
$$ LANGUAGE plpgsql;
```

```
cursor.execute("SELECT * FROM buscar_productos(%s, %s)", (categoria, precio_max))
```

Prevención: Principio de mínimo privilegio

El principio de mínimo privilegio es la segunda línea de defensa.

Incluso si un atacante logra inyectar SQL, sus acciones están limitadas por los permisos del rol de la aplicación.

El rol de la aplicación debe ser restrictivo

```
-- Rol para la aplicación web de ventas
CREATE ROLE app_tienda LOGIN PASSWORD 'clave_segura';

GRANT CONNECT ON DATABASE tienda TO app_tienda;
GRANT USAGE ON SCHEMA public TO app_tienda;

-- Solo lo que la aplicación necesita
GRANT SELECT ON TABLE productos, categorias TO app_tienda;
GRANT SELECT, INSERT ON TABLE pedidos, pedido_detalle TO app_tienda;
GRANT SELECT ON TABLE clientes TO app_tienda;
GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA public TO app_tienda;

-- Explícitamente: sin acceso a datos sensibles
-- NO se le da acceso a: usuarios, salarios, logs internos
```

El impacto del mínimo privilegio

Sin mínimo privilegio (app conectada como superusuario):

```
-- El atacante puede hacer todo
' UNION SELECT username, password FROM usuarios --
'; DROP TABLE pedidos; --
'; CREATE ROLE hacker SUPERUSER LOGIN PASSWORD 'pwned'; --
```

Con mínimo privilegio (app con permisos limitados):

```
-- El atacante intenta
' UNION SELECT username, password FROM usuarios --
-- ERROR: permission denied for table usuarios

'; DROP TABLE pedidos; --
-- ERROR: permission denied for table pedidos (no tiene DELETE)
```

La inyección sigue siendo posible, pero el daño está contenido.

Separar roles por función

```
-- Un rol solo para lectura (reportes)
CREATE ROLE app_reportes LOGIN PASSWORD 'rpt_clave';
GRANT SELECT ON ALL TABLES IN SCHEMA public TO app_reportes;

-- Un rol para operaciones de escritura (la app web)
CREATE ROLE app_web LOGIN PASSWORD 'web_clave';
GRANT SELECT ON TABLE productos, clientes TO app_web;
GRANT INSERT ON TABLE pedidos TO app_web;

-- Un rol para administración (solo humanos, no apps)
CREATE ROLE admin_db LOGIN PASSWORD 'adm_clave' CREATEROLE;
```

Si la app web es comprometida, el atacante no puede leer datos de reportes ni administrar roles.

Prevención: Validación de entrada

La validación de entrada **no reemplaza** las consultas parametrizadas.

Es una capa adicional de defensa.

Validar el tipo de dato

```
# Si se espera un entero, verificar que lo sea
def obtener_producto(producto_id):
    if not str(producto_id).isdigit():
        raise ValueError("ID de producto inválido")

    cursor.execute("SELECT * FROM productos WHERE id = %s", (producto_id,))
```

Validar con listas blancas

Para valores que deben ser exactamente uno de un conjunto conocido:

```
CATEGORIAS_VALIDAS = {'electronica', 'ropa', 'alimentos', 'libros'}

def buscar_por_categoria(categoria):
    if categoria not in CATEGORIAS_VALIDAS:
        raise ValueError("Categoría no válida")

    cursor.execute(
        "SELECT * FROM productos WHERE categoria = %s",
        (categoria,)
    )
```

Sanitizar no es suficiente

Intentar "escapar" o "limpiar" el input manualmente es propenso a errores:

```
# Esto NO es seguro — hay formas de evadirlo
username = username.replace("'", "''")
query = "SELECT * FROM usuarios WHERE username = '" + username + "'"
```

Siempre usar consultas parametrizadas. La validación es complementaria, no sustituta.

Errores comunes

Error 1: concatenar strings para construir SQL

```
# Vulnerable — siempre
query = "SELECT * FROM " + tabla + " WHERE id = " + id
cursor.execute(query)
```

No hay forma segura de construir SQL por concatenación cuando los datos vienen del usuario.

Error 2: confiar en que el frontend valida

Un atacante no usa el formulario web.

Envía peticiones HTTP directamente con herramientas como `curl` o Burp Suite.

La validación del frontend es solo experiencia de usuario, no seguridad.

Error 3: mostrar mensajes de error de la base de datos

```
ERROR: syntax error at or near "'" LINE 1: SELECT * FROM usuarios WHERE username = 'admin
```

Los mensajes de error revelan la estructura de la consulta.

El atacante usa esa información para ajustar su ataque.

En producción: capturar errores internamente, mostrar mensajes genéricos al usuario.

Error 4: usar el superusuario en aplicaciones

Si la aplicación se conecta con `postgres` o un rol con `SUPERUSER`:

Un ataque exitoso tiene acceso ilimitado a toda la base de datos.

Crear siempre un rol específico con mínimo privilegio.

Error 5: creer que ORM protege automáticamente

Los ORM protegen cuando se usan correctamente.

Pero permiten consultas crudas, y ahí puede aparecer la vulnerabilidad:

```
# Django ORM — seguro
User.objects.filter(username=username)

# Django ORM — vulnerable (raw query con formato)
User.objects.raw(f"SELECT * FROM users WHERE username = '{username}'")
```

Defensa en profundidad

Ninguna medida por sí sola es suficiente.

La seguridad real combina múltiples capas.

Las capas de defensa

| Capa | Medida | Qué previene | | --- | --- | --- | | Código | Consultas parametrizadas | Inyección de SQL | | Código | Validación de entrada | Inputs malformados | | Base de datos | Mínimo privilegio | Daño si hay inyección | | Base de datos | Roles separados por función | Escalada lateral | | Aplicación | Mensajes de error genéricos | Reconocimiento del atacante | | Infraestructura | Firewall de aplicación web (WAF) | Ataques conocidos | | Monitoreo | Logs de consultas lentas o inusuales | Detección de ataques en curso |

El principio

Asumir que cada capa puede fallar.

Diseñar el sistema para que, si falla una capa, las demás contengan el daño.

¿Qué puede salir mal?

- **Consultas por concatenación** → vulnerabilidad directa a SQL Injection
- **Rol de la aplicación con ALL PRIVILEGES** → daño ilimitado si hay inyección exitosa
- **Contraseñas almacenadas en texto plano** → inyección expone credenciales reales
- **Mensajes de error técnicos al usuario** → el atacante mapea la estructura de la base de datos
- **Validar solo en el frontend** → ignorado trivialmente con cualquier cliente HTTP
- **Usar el mismo rol para lectura y escritura cuando no es necesario** → innecesariamente amplio
- **Creer que el ORM es inmune** → las raw queries dentro del ORM son igual de vulnerables
- **No registrar consultas en logs** → los ataques pasan desapercibidos