
# INSTITUTO POLITÉCNICO NACIONAL

# ESCUELA SUPERIOR DE CÓMPUTO

## BASES DE DATOS

GRUPO: 3CV5

Profesor: Gabriel Hurtado Avilés

# “Práctica 4.1 Seguridad y Vulnerabilidades: SQL Injection”

## “Sillas y Mesas Hernández”

**Alumno**:

- González González Erick Emiliano
- De La Rosa Hernández Tania

Fecha de Entrega: 14/10/25

# 1: INVESTIGACIÓN Y ANÁLISIS

## A) ¿Qué es SQL Injection?

Definición técnica:

SQL Injection (SQLi) es una vulnerabilidad que ocurre cuando una aplicación incorpora datos no confiables (por ejemplo, la entrada del usuario) dentro de consultas SQL sin validación o sanitización adecuada. Esto permite que un atacante altere la estructura de la consulta y ejecute comandos SQL arbitrarios, como leer, modificar o eliminar datos en la base de datos.

Tipos principales:

Union-based (in-band, UNION): El atacante usa la cláusula UNION SELECT para unir los resultados de una consulta maliciosa con la legítima y así obtener información de otras tablas.

Boolean-based (blind): El atacante envía condiciones que devuelven respuestas verdaderas o falsas. A partir de las diferencias en la respuesta, puede deducir información de la base de datos.

Time-based (blind): El atacante usa funciones que retrasan la respuesta del servidor, como SLEEP(n), para inferir si una condición es verdadera observando el tiempo que tarda en responder.

Impacto en seguridad (OWASP Top 10):

SQL Injection se considera una de las vulnerabilidades más graves en el desarrollo web, ya que permite el acceso no autorizado a información sensible e incluso la toma total del control del sistema. OWASP la incluye constantemente entre las principales amenazas a la seguridad de aplicaciones web.

## B) Análisis del código vulnerable

```php
<?php
$username = $_POST['username'];
$password = $_POST['password'];

$sql = "SELECT id, username FROM users WHERE username = '" . $username . "' AND password = '" . $password . "'";
$result = $conn->query($sql);
?>
```

La consulta se construye concatenando directamente la entrada del usuario ($_POST), por lo que un atacante puede insertar comandos SQL maliciosos que el servidor ejecutará.

La vulnerabilidad se atribuye a que los datos del usuario no se tratan como parámetros sino como parte del código SQL. Entonces, el atacante puede modificar la estructura de la consulta al inyectar comillas, operadores lógicos o comentarios.

Ejemplo de payload de ataque:


Y como consulta resultante, el atacante recibirá:


Así, el resto del código quedará comentado y se podrá acceder sin contraseña

## C) ORM como medida de prevención

¿Qué es un ORM?
Un Object-Relational Mapping (ORM) es una técnica que permite interactuar con la base de datos usando objetos del lenguaje de programación, en lugar de escribir consultas SQL manualmente. Así, las tablas y filas se representan como clases y objetos.

¿Cómo previene SQL Injection?
Los ORMs internamente utilizan consultas parametrizadas y envían los valores del usuario por separado del código SQL. De esta forma, el motor de la base de datos interpreta los datos como valores, no como instrucciones ejecutables, evitando que se modifique la estructura de la consulta.

Investigación sobre GORM (Go):
GORM es un ORM muy utilizado en el lenguaje Go. Su sistema de consultas utiliza placeholders (?) para los parámetros, lo que protege de inyecciones si se usa correctamente.
Por ejemplo: *db.Where("email = ?", email).First(&user)*

Esta instrucción es segura porque email se envía como un parámetro, no como parte del SQL. Sin embargo, si se usan consultas *raw* (SQL escrito directamente) con concatenación de cadenas, el riesgo de SQL Injection puede reaparecer.

Ventajas y limitaciones:

- Ventajas**:**
  - Evita la concatenación manual de strings.
  - Usa parámetros automáticamente.
  - Mejora la legibilidad y mantenimiento del código.
- Limitaciones**:**
  - Si se usa SQL crudo con concatenación, puede ser vulnerable.
  - No sustituye controles de acceso ni validaciones de entrada.
  - Puede generar un pequeño impacto en rendimiento comparado con SQL directo.

## D) Relación con DCL (Data Control Language)

¿Por qué los permisos de BD no detienen SQL Injection?

Los permisos en la base de datos (como GRANT o REVOKE) controlan lo que puede hacer un usuario autenticado. Sin embargo, si la aplicación se conecta a la base de datos con una cuenta que tiene permisos amplios (por ejemplo, lectura y escritura), un atacante que explote una inyección actuará con esos mismos privilegios. Por eso los permisos no detienen la vulnerabilidad.

Capas de seguridad necesarias:

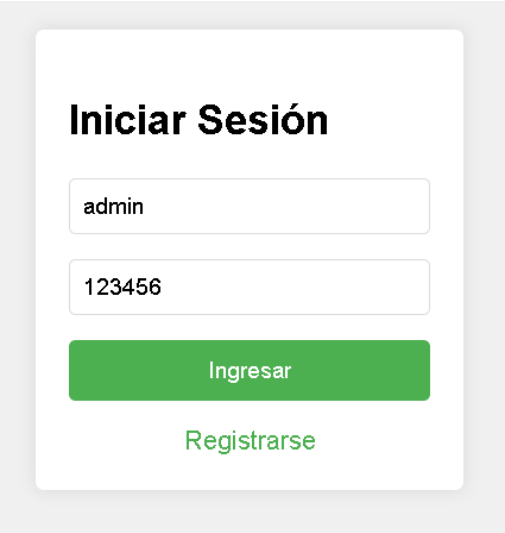
- Uso de prepared statements o consultas parametrizadas.
- Validación y sanitización de entradas.
- Cuentas de base de datos con privilegios mínimos.
- Monitoreo de actividad y registros de auditoría.
- Uso de cortafuegos de aplicaciones web (WAF).

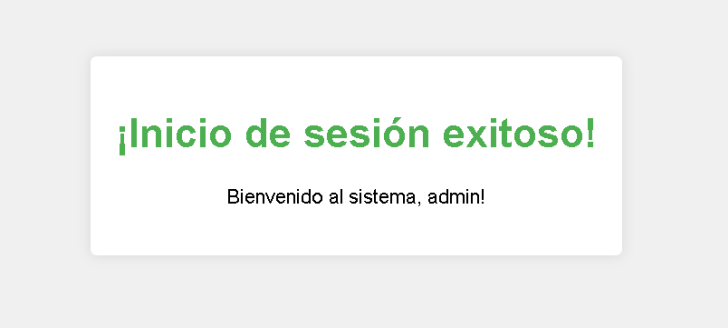
Principio de defensa en profundidad:

Este principio establece que ninguna capa de seguridad es suficiente por sí sola. La protección contra ataques como SQL Injection debe basarse en múltiples capas: código seguro, configuraciones adecuadas de la base de datos, permisos restringidos, validación de entrada y monitoreo constante. Si una capa falla, las demás reducen el impacto o impiden que el ataque tenga éxito.

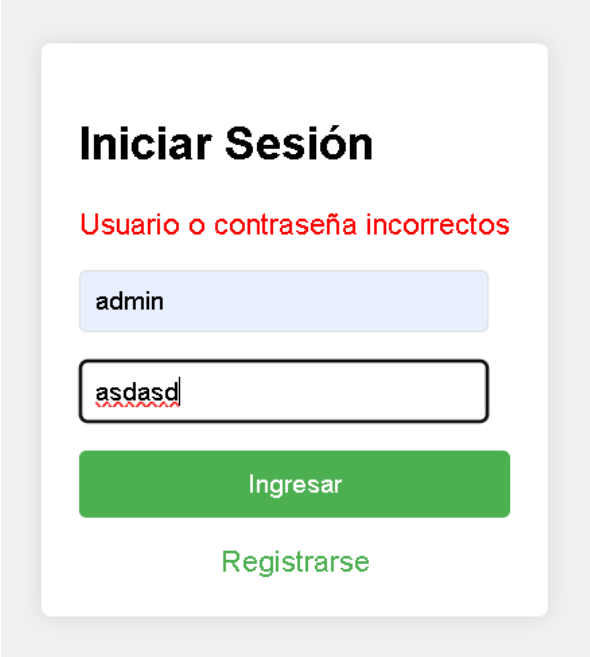
# 2: IMPLEMENTACIÓN

Módulo de inicio de sesión

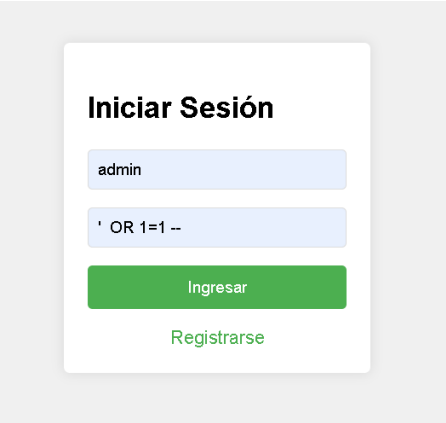


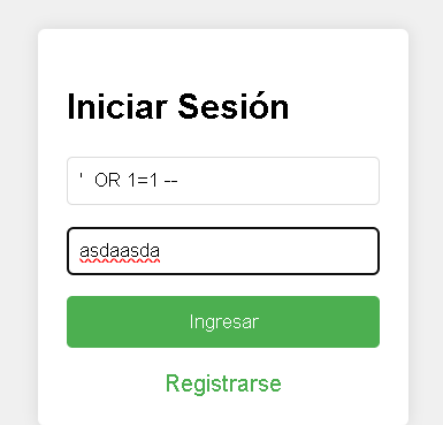

1. Intento de inicio de sesión con credenciales incorrectas

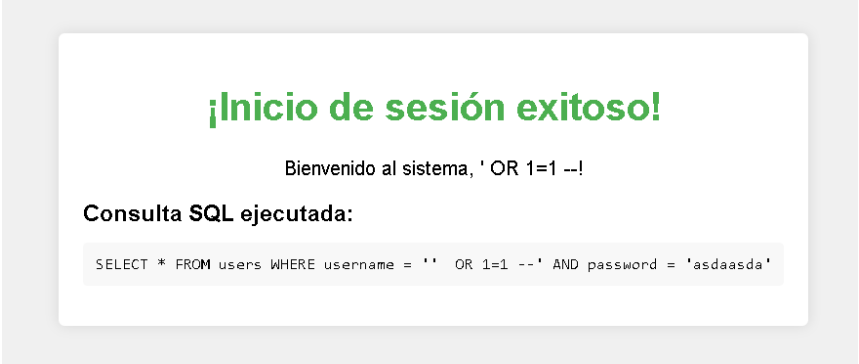


2. Intento de inicio de sesión con inyección SQL









3. Fragmentos de código **vulnerable**

```python
# Vulnerable: construye la consulta con f-string (o concatenación)
username = request.form.get('username')
password = request.form.get('password')

query = f"SELECT * FROM users WHERE username = '{username}' AND
password = '{password}'"
c.execute(query)
user = c.fetchone()
```

4. Fragmento de código con uso de **parámetros**

```
# Protegido: consulta parametrizada (sqlite3)
username = request.form.get('username')
password = request.form.get('password')

c.execute('SELECT * FROM users WHERE username = ? AND password =
?', (username, password))
user = c.fetchone()
```

5. Uso adicional de contraseñas **hasheadas**

```
from werkzeug.security import generate_password_hash

username = request.form.get('username')
password = request.form.get('password')
hashed = generate_password_hash(password)  # bcrypt/sha adaptado
por werkzeug

c.execute('INSERT INTO users (username, password) VALUES (?, ?)',
(username, hashed))
conn.commit()
```

6. Login seguro

```
from werkzeug.security import check_password_hash

username = request.form.get('username')
password = request.form.get('password')

c.execute('SELECT id, username, password FROM users WHERE username
= ?', (username,))
row = c.fetchone()
if row and check_password_hash(row[2], password):
    # autenticado
    ...
else:
    # fallo
    ...
```

# Conclusiones

Durante esta práctica comprendimos la importancia de aplicar medidas básicas de seguridad en bases de datos. Pudimos ver cómo una aplicación puede ser vulnerable si no se valida correctamente la información que envía el usuario, y cómo una consulta mal construida puede permitir accesos no autorizados.

También nos percatamos de que almacenar contraseñas en texto plano es un gran riesgo, y que usar funciones como password_hash() y password_verify() permite proteger mejor la información de los usuarios.

La práctica mostró claramente la diferencia entre usar una consulta insegura (concatenando texto) y una segura (con sentencias preparadas). Esto nos ayudó a entender cómo funcionan las inyecciones SQL y cómo prevenirlas desde el código.

Finalmente, entendimos que la seguridad no depende solo de la base de datos, sino también de cómo se diseñan las aplicaciones que la usan. Aunque parezcan detalles pequeños, seguir buenas prácticas como validar entradas, usar hashes y restringir permisos hace una gran diferencia para proteger los datos.

# Bibliografía

*A03 injection - OWASP top 10:2021*. (s/f). Owasp.org. Recuperado el 15 de octubre de 2025, de https://owasp.org/Top10/A03_2021-Injection/

Jinzhu. (2025, octubre 14). *Security*. GORM. https://gorm.io/docs/security.html

Radware. (s/f). *SQL injection: Examples, real life attacks & 9 defensive measures*. Radware.com. Recuperado el 15 de octubre de 2025, de https://www.radware.com/cyberpedia/application-security/sql-injection/

*Types of SQL injection?* (s/f). Acunetix. Recuperado el 15 de octubre de 2025, de https://www.acunetix.com/websitesecurity/sql-injection2/

*What is SQL Injection? Tutorial & Examples*. (s/f). Portswigger.net. Recuperado el 15 de octubre de 2025, de https://portswigger.net/web-security/sql-injection