



# Programación en Bases de Datos

---

## Proyecto Hacking Ético: SQL Injection



## Índice de contenidos

<b>1. Introducción al proyecto</b>	<b>2</b>
<b>2. Inyección SQL</b>	<b>3</b>
2.1. ¿Qué es una inyección SQL? . . . . .	3
2.2. Causa principal de las inyecciones SQL . . . . .	4
2.3. Tipos de inyecciones SQL . . . . .	6
<b>3. Preparación del entorno</b>	<b>7</b>
3.1. Prerequisitos . . . . .	7
3.2. Requerimientos de ejecución . . . . .	7
<b>4. Conexión de la base de datos</b>	<b>7</b>
4.1. Oracle . . . . .	8
4.2. PostgreSQL . . . . .	8
<b>5. Creación de tablas</b>	<b>9</b>
5.1. Oracle . . . . .	9
5.2. PostgreSQL . . . . .	11
<b>6. Introducción al laboratorio</b>	<b>12</b>
<b>7. Inyección basada en errores de la base de datos</b>	<b>12</b>
7.1. Mecanismo del Ataque . . . . .	12
7.2. Login sin credenciales válidas . . . . .	13
7.3. información sobre las tablas . . . . .	15
7.4. Código vulnerable del login . . . . .	15
<b>8. Inyección basada en Union Attack</b>	<b>15</b>
8.1. Mecanismo del ataque . . . . .	16
8.2. Obtención de nombre de la base de datos . . . . .	17
8.2.1. Explicación del funcionamiento . . . . .	17
8.3. Obtención de la versión de la base de datos . . . . .	18
8.3.1. Explicación del funcionamiento . . . . .	19
8.4. Obtención de todas las tablas de la base de datos . . . . .	19
8.4.1. Explicación del funcionamiento . . . . .	20

## 1. Introducción al proyecto

El presente documento aborda el tema del hacking ético y su aplicación en el estudio de vulnerabilidades en bases de datos, con un enfoque particular en las inyecciones SQL. Estas vulnerabilidades constituyen uno de los riesgos más comunes y críticos en aplicaciones web que interactúan con bases de datos, permitiendo a los atacantes manipular o acceder a información de manera no autorizada. Este trabajo tiene como propósito profundizar en la comprensión de las inyecciones SQL, analizando sus tipos, los riesgos asociados y los motivos por los cuales ocurren, para fomentar una perspectiva integral de la seguridad en el desarrollo de software. Para facilitar el aprendizaje práctico y la evaluación de estas vulnerabilidades, se ha desarro-

llado un laboratorio interactivo basado en tecnologías como Flask y Python. Este entorno de simulación permite experimentar con diferentes tipos de inyecciones SQL, ofreciendo a los usuarios una experiencia inmersiva y controlada. El laboratorio incluye aplicaciones web diseñadas específicamente para emular escenarios reales de inyecciones SQL, permitiendo realizar pruebas tanto en bases de datos Oracle como en PostgreSQL. Estas bases de datos, seleccionadas por su relevancia y amplio uso en entornos empresariales, proporcionan un contexto diverso y representativo para explorar las técnicas de ataque y sus consecuencias.

Cada tipo de inyección SQL implementada en el laboratorio ha sido cuidadosamente seleccionada para cubrir un amplio espectro de vulnerabilidades. Entre ellas, se encuentran las inyecciones basadas en errores, que explotan los mensajes de error generados por la base de datos para extraer información sensible; las inyecciones de tipo "Union Attack", que utilizan la cláusula UNION para combinar resultados de consultas y obtener datos confidenciales; las inyecciones basadas en booleanos, que permiten inferir información mediante la manipulación de condiciones lógicas; y las inyecciones ciegas, que aprovechan diferencias en el comportamiento de la aplicación para deducir detalles de la base de datos sin generar mensajes de error explícitos.

El diseño del laboratorio incluye páginas individuales para cada tipo de inyección, con formularios de inicio de sesión vulnerables, credenciales específicas para realizar las pruebas y explicaciones detalladas sobre el ataque correspondiente. Esto permite que los usuarios comprendan tanto la teoría subyacente como la ejecución práctica de cada tipo de inyección. Además, el laboratorio cuenta con documentación complementaria que explica cómo se implementaron las vulnerabilidades y su relevancia en entornos reales, proporcionando una base sólida para que los participantes puedan identificar y mitigar estos riesgos en sus propios proyectos.

Este trabajo no solo subraya los riesgos asociados con las inyecciones SQL, como el acceso no autorizado a datos confidenciales, la alteración o eliminación de información crítica y el compromiso total del servidor de base de datos, sino que también explora las causas más comunes detrás de estas vulnerabilidades. Entre ellas, se encuentran la falta de sanitización de las entradas del usuario, el uso de consultas dinámicas inseguras y la ausencia de auditorías de seguridad durante el desarrollo de software. Al comprender estos factores, se busca fomentar la adopción de mejores prácticas en la construcción de aplicaciones seguras y resilientes.

En conclusión, este proyecto combina un enfoque teórico y práctico para abordar una de las amenazas más relevantes en el ámbito de la seguridad informática. Al proporcionar un entorno interactivo y educativo, se espera que este laboratorio no solo aumente la conciencia sobre la importancia de prevenir las inyecciones SQL, sino que también capacite a los participantes en la identificación, explotación controlada y mitigación de estas vulnerabilidades. Este esfuerzo

refleja el compromiso con la promoción de un desarrollo de software más seguro, alineado con los principios del hacking ético y la ciberseguridad.

## 2. Inyección SQL

### 2.1. ¿Qué es una inyección SQL?

Imagina que estás en un restaurante y entregas tu orden al mesero. Si todo funciona como debería, el mesero simplemente transmite tu pedido al chef, quien prepara la comida según lo solicitado. Ahora bien, ¿qué pasaría si, en lugar de un pedido normal, decides escribir en la nota algo como: "Quiero una pizza, y además dame acceso a la caja registradora? Si el sistema del restaurante no tiene medidas para validar las órdenes, es posible que el mensaje llegue al chef, quien podría malinterpretarlo como una instrucción válida y permitirte hacer algo que no deberías poder hacer. Algo similar ocurre con una inyección SQL, pero en el contexto de aplicaciones web y bases de datos.

Una inyección SQL es un tipo de ataque en el que un atacante manipula las consultas que una aplicación web hace a su base de datos para ejecutar comandos maliciosos. Esto sucede cuando el sistema no valida adecuadamente las entradas proporcionadas por los usuarios y las trata directamente como parte de la consulta SQL. Por ejemplo, en una página de inicio de sesión, si un usuario malintencionado introduce un texto diseñado específicamente para alterar la lógica de la consulta SQL que valida las credenciales, podría obtener acceso no autorizado al sistema sin necesidad de conocer la contraseña.

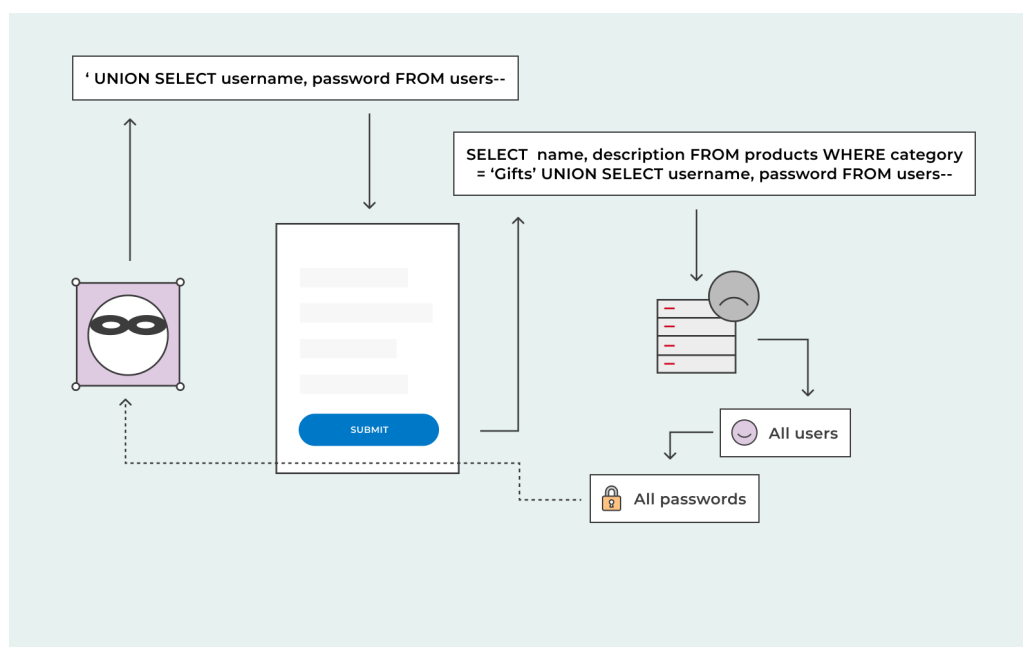


Figura 1: Ilustración del flujo de una SQLI

Un caso real que ilustra el impacto de las inyecciones SQL ocurrió en el año 2012, cuando un grupo de atacantes explotó esta vulnerabilidad en una aplicación de LinkedIn. Mediante una inyección SQL, los atacantes lograron acceder a información confidencial de usuarios, incluidos datos de inicio de sesión. Este incidente no solo expuso la información personal de millones

de personas, sino que también dañó la reputación de la compañía y resaltó la gravedad de no implementar medidas de seguridad adecuadas.

Por ejemplo, imagina una consulta SQL en una aplicación web que maneja la autenticación de usuarios. Una consulta típica podría ser:

```
1 SELECT * FROM usuarios
2 WHERE nombre_usuario = 'usuario'
3 AND contraseña = 'contraseña';
```

Si el sistema permite que un atacante ingrese algo como `usuario' OR '1'='1` en lugar del nombre de usuario, la consulta resultante sería:

```
1 SELECT * FROM usuarios
2 WHERE nombre_usuario = 'usuario' OR '1'='1'
3 AND contraseña = 'contraseña';
```

La condición `'1'='1'` siempre es verdadera, lo que significa que el atacante podría acceder al sistema sin necesidad de proporcionar credenciales válidas. Este ejemplo muestra cómo una entrada maliciosa puede alterar la lógica de una consulta SQL, dándole al atacante el control.

Las inyecciones SQL pueden tener consecuencias graves, que incluyen el robo de datos confidenciales, la modificación o eliminación de registros, y en casos extremos, el control total del servidor de la base de datos. Además, son una de las vulnerabilidades más comunes en aplicaciones web, según el *OWASP* (Open Web Application Security Project). Sin embargo, prevenirlas no es complicado si se aplican prácticas adecuadas, como el uso de consultas preparadas, la validación estricta de entradas y la implementación de controles de acceso robustos.

En resumen, una inyección SQL no es solo un fallo técnico, sino un ejemplo de cómo pequeñas omisiones en la seguridad pueden tener grandes repercusiones. Conocer cómo ocurren y cómo prevenirlas es esencial para cualquier profesional que desarrolle aplicaciones conectadas a bases de datos, ya que no solo se trata de proteger sistemas, sino también la confianza y la información de los usuarios.

## 2.2. Causa principal de las inyecciones SQL

La principal causa de las inyecciones SQL es una mala validación de las entradas proporcionadas por los usuarios. Esto ocurre, en muchos casos, debido a la concatenación directa de parámetros en las consultas SQL, lo que permite que un atacante manipule la estructura de la consulta para incluir comandos maliciosos. Este enfoque no solo es inseguro, sino que también facilita la explotación de las vulnerabilidades al no distinguir entre los datos del usuario y el código SQL.

Un ejemplo típico de concatenación de parámetros puede observarse en la siguiente función de autenticación insegura para una base de datos Oracle. Aquí, los valores proporcionados por el usuario, como el nombre de usuario y la contraseña, se concatenan directamente en la consulta:

```
1 # Función de autenticación insegura
2 def login_inseguro_base_oracle(username, password):
3     .
```

```
4 .
5 .
6 .
7 .
8 sentencia = "SELECT * FROM Usuarios WHERE username = '"+username+
9             "' AND password = '"+password+"'"
10
11
12
13
14 try:
15     cursor = conexion.cursor()
16     cursor.execute(sentencia)
17     usuario = cursor.fetchall()
18     cursor.close()
19     if usuario:
20         print("Usuario autenticado:", usuario)
21         return {"resultado":usuario, "sentencia":sentencia, "auth":
22                 "true"}
23     else:
24
25 .
```

En este ejemplo, si un atacante introduce un valor malicioso como `username = 'admin'` en lugar de un nombre de usuario legítimo, puede manipular la consulta para obtener acceso al sistema sin necesidad de una contraseña válida. Esto ocurre porque la concatenación no distingue entre datos y comandos SQL, permitiendo al atacante alterar la lógica de la consulta.

Una forma segura de evitar este tipo de vulnerabilidades es utilizar la vinculación de parámetros en lugar de la concatenación. Este enfoque asegura que los datos del usuario sean tratados exclusivamente como valores y no como parte del código SQL. A continuación, se presenta un ejemplo seguro de autenticación utilizando vinculación de parámetros en Oracle:

```
1 # Función de autenticación segura
2 def login_seguro_oracle(username, password):
3 .
4 .
5 .
6 .
7 try:
8     cursor = conexion.cursor()
9     cursor.execute("SELECT * FROM Usuarios
10     WHERE username = :user
11     AND password = :pass", user=username, pass=password)
12     usuario = cursor.fetchone()
13     cursor.close()
14     if usuario:
15         print("Usuario autenticado:", usuario)
```

```
16         return True
17     else:
18 .
19 .
20 .
```

En este caso, el uso de `:user` y `:pass` como marcadores de posición permite que los valores del usuario sean procesados de manera segura por el motor de la base de datos. Esto elimina cualquier posibilidad de que se interpreten como comandos SQL, previniendo ataques de inyección SQL. Este enfoque no solo mejora la seguridad de la aplicación, sino que también fomenta mejores prácticas en el manejo de entradas de usuario.

## 2.3. Tipos de inyecciones SQL

Las inyecciones SQL pueden manifestarse de diferentes formas, dependiendo de la vulnerabilidad específica que se explote y del comportamiento del sistema. A continuación, se presentan algunos de los tipos más comunes de inyecciones SQL y cómo se manifiestan en el contexto de una aplicación web.

- **Inyección basada en errores:** Este tipo de inyección aprovecha los mensajes de error generados por la base de datos para obtener información sobre la estructura y el contenido de la base de datos. Al introducir consultas mal formadas, un atacante puede provocar errores que revelan detalles sensibles, como nombres de tablas o columnas.
- **Inyección de tipo "Union Attack":** Las inyecciones de tipo "Union Attack" se basan en la cláusula UNION de SQL para combinar resultados de consultas y obtener información confidencial. Al manipular las consultas para incluir una instrucción UNION, un atacante puede extraer datos de tablas no autorizadas.
- **Inyección basada en booleanos:** Las inyecciones basadas en booleanos se aprovechan de las diferencias en el comportamiento de la aplicación para inferir información sobre la base de datos. Al modificar las condiciones lógicas de las consultas, un atacante puede deducir detalles sobre la estructura y el contenido de la base de datos.
- **Inyección blind:** Las inyecciones ciegas se caracterizan por la falta de mensajes de error explícitos, lo que dificulta la identificación de la vulnerabilidad. Al manipular las consultas para observar cambios en el comportamiento de la aplicación, un atacante puede inferir información sobre la base de datos sin generar alertas.
- **Inyección blind de tiempo:** Las inyecciones de tiempo se basan en la introducción de retrasos deliberados en las consultas para inferir información sobre la base de datos. Al introducir instrucciones que causan demoras en la respuesta, un atacante puede deducir detalles sobre la estructura y el contenido de la base de datos.

Cada tipo de inyección SQL presenta desafíos y riesgos únicos, que van desde la exposición de información confidencial hasta la alteración de registros críticos. Al comprender cómo ocurren y cómo prevenirlas, los desarrolladores y profesionales de la seguridad pueden fortalecer la protección de sus aplicaciones y bases de datos, reduciendo así la exposición a riesgos innecesarios.

### 3. Preparación del entorno

En este apartado se detallarán los requerimientos necesarios para la correcta ejecución del proyecto, así como las funciones que se han implementado para la creación de la base de datos y la inserción de datos en la misma.

#### 3.1. Prerequisitos

A modo de base para el correcto desarrollo y ejecución del proyecto, es necesario tener instalado en el sistema los siguientes paquetes:

- Oracle Database
- PostgreSQL Database
- Python

La version descargada de los anteriores paquetes puede ser la que ha sido configurada en anteriores prácticas desarrolladas en la asignatura de Programación en Bases de Datos.

#### 3.2. Requerimientos de ejecución

Una vez configurados correctamente los prerequisites que no estan ligados especificamente con esta práctica, se procederá a la instalación de las librerías y frameworks de python que han sido necesarios para el desarrollo de este laboratorio. Para facilitar este proceso se ha generado un fichero con las librerías necesarias que se puede instalar mediante el siguiente comando:

```
1 pip install -r requirements.txt
```

En dicho fichero *requirements.txt* se encuentran las siguientes librerías:

- **Flask** con version 2.2.0
- **werkzeug** con version 2.2.0
- **oracledb** con version mayor o igual a 2.4.1
- **psycopg2** con version mayor o igual a 2.9.9
- **requests** con version mayor o igual a 2.32.2
- **termcolor** con version 2.2.0
- **yaspin** con version mayor o igual a 3.1.0

### 4. Conexión de la base de datos

Para la conexión de la base de datos se han implementado funciones de conectar y desconectar que permiten la conexión a Oracle y PostgreSQL. Estas funciones se han implementado en los archivos *setupOracle.py* y *setupPostgres.py*. A continuación se detallan las funciones implementadas en cada uno de los archivos.



## 4.1. Oracle

A continuación se muestran las funciones que se ha implementado en el archivo *setupOracle.py* para la conexión a Oracle.

```
1 # Función para conectar a la base de datos
2 def dbConectarOracle():
3     ip = "localhost"
4     puerto = 1521
5     s_id = "xe"
6     usuario = "system"
7     contrasena = "12345"
8
9     print("---dbConectarOracle---")
10    print("---Conectando a Oracle---")
11
12    try:
13        conexion = PBD.connect(user=usuario, password=contrasena, host=
14                                ip, port=puerto, sid=s_id)
15        print("Conexión realizada a la base de datos", conexion)
16        return conexion
17    except PBD.DatabaseError as error:
18        print("Error en la conexión")
19        print(error)
20        return None
21
22 # Función para desconectar de la base de datos
23 def dbDesconectar(conexion):
24     print("---dbDesconectar---")
25     try:
26         if conexion: # Verifica que la conexión no sea None
27             conexion.commit() # Confirma los cambios
28
29             conexion.close()
30             print("Desconexión realizada correctamente")
31             return True
32         else:
33             print("No hay conexión para cerrar.")
34             return False
35     except PBD.DatabaseError as error:
36         print("Error en la desconexión")
37         print(error)
38         return False
```

## 4.2. PostgreSQL

A continuación se muestra la funciones que se ha implementado en el archivo *setupPostgres.py* para la conexión a PostgreSQL.

```
1
2 def dbConectarPostgreSQL():
3     ip = "localhost"
4     puerto = 5432
5     basedatos = "Empresa"
```

```
6
7 usuario = "postgres"
8 contrasena = "12345"
9
10 print("---dbConectarPostgreSQL---")
11 print("---Conectando a Postgresql---")
12
13 try:
14     conexion = PBD.connect(user=usuario, password=contrasena, host=ip,
15                             port=puerto, database=basedatos)
16     print("Conexión realizada a la base de datos",conexion)
17     return conexion
18 except PBD.DatabaseError as error:
19     print("Error en la conexión")
20     print(error)
21     return None
22
23 # -----
24
25 def dbDesconectar(conexion):
26     print("---dbDesconectar---")
27     try:
28         conexion.commit() # Confirma los cambios
29         conexion.close()
30         print("Desconexión realizada correctamente")
31         return True
32     except PBD.DatabaseError as error:
33         print("Error en la desconexión")
34         print(error)
35         return False
```

## 5. Creación de tablas

Para la creación de las tablas en Oracle y PostgreSQL se han implementado dos funciones que permiten la creación de las mismas. Se ha decidido crear una tabla llamada *Usuarios* con los campos *id*, *username*, *password* y *session\_cookie*. Además, se ha incluido la inserción de usuarios de ejemplo en la tabla, para poder hacer uso de ellos en las inyecciones del laboratorio. Estas funciones se han implementado en los archivos *setupOracle.py* y *setupPostgres.py*. A continuación se detallan las funciones implementadas en cada uno de los archivos.

### 5.1. Oracle

A continuación se muestra la función que se ha implementado en el archivo *setupOracle.py* para la creación de la tabla en Oracle.

```
1 # Funcion para la configuracion de tablas
2 def configuracionTablas\_oracle(conexion):
3     print("---configuracionTablas---")
4     try:
5         cursor = conexion.cursor()
6
```

```
7      # Crear tabla Usuarios si no existe con columna session\
      _cookie
8      consulta = """
9          BEGIN
10             EXECUTE IMMEDIATE 'CREATE TABLE Usuarios (
11                 id NUMBER GENERATED BY DEFAULT AS IDENTITY
12                     PRIMARY KEY,
13                 username VARCHAR2(50) NOT NULL UNIQUE,
14                 password VARCHAR2(50) NOT NULL,
15                 session\_cookie VARCHAR2(255)
16             )';
17         EXCEPTION
18             WHEN OTHERS THEN
19                 IF SQLCODE = -955 THEN
20                     NULL; -- Ignora si la tabla ya existe
21                 ELSE
22                     RAISE;
23                 END IF;
24         END;
25     """
26     cursor.execute(consulta)
27
28     # Insertar usuarios de ejemplo solo si la tabla esta vacia
29     cursor.execute("SELECT COUNT(*) FROM Usuarios")
30     count = cursor.fetchone()[0]
31     print("Usuarios en la tabla:", count)
32     if count == 0:
33         usuarios\_ejemplo = [
34             ("admin", "password123", "
35             t4SpnpWyg76A3K2BqcFh2vODqOfqJGvs38ydh9"),
36             ("user1", "password1", "
37             d382yd8n21df4314fn817yf6834188ls023d8d"),
38             ("user2", "password2", "
39             u73dv226d726gh23fnjncuyg0q9udfjf47eueu")
40         ]
41         cursor.executemany(
42             "INSERT INTO Usuarios (username, password, session\
43             _cookie) VALUES (:username, :password, :session\
44             _cookie)",
45             usuarios\_ejemplo
46         )
47         print("Usuarios de ejemplo insertados correctamente.")
48     else:
49         print("La tabla Usuarios ya contiene datos.")
50
51     cursor.close()
52     print("Tabla 'Usuarios' creada o verificada exitosamente")
53     return True
54 except PBD.DatabaseError as error:
55     print("Error al crear la tabla o insertar usuarios")
56     print(error)
57     return False
```

## 5.2. PostgreSQL

A continuación se muestra la función que se ha implementado en el archivo *setupPostgres.py* para la creación de la tabla en PostgreSQL.

```
1 def configuracion_tablas_postgresql(conexion):
2     print("---configuracion_tablas_postgresql---")
3     try:
4         cursor = conexion.cursor()
5
6         # Crear la tabla Usuarios si no existe con columna session_cookie
7         consulta = """
8             CREATE TABLE IF NOT EXISTS Usuarios (
9                 id SERIAL PRIMARY KEY,
10                username VARCHAR(50) NOT NULL UNIQUE,
11                password VARCHAR(50) NOT NULL,
12                session_cookie VARCHAR(255)
13            );
14        """
15        cursor.execute(consulta)
16
17        # Insertar usuarios de ejemplo solo si la tabla esta vacia
18        cursor.execute("SELECT COUNT(*) FROM Usuarios")
19        count = cursor.fetchone()[0]
20        if count == 0:
21            usuarios_ejemplo = [
22                ("admin", "password123", "
23                    t4SpnpWyg76A3K2BqcFh2v0Dq0fqJGvs38ydh9"),
24                ("user1", "password1", "
25                    d382yd8n21df4314fn817yf6834188ls023d8d"),
26                ("user2", "password2", "
27                    u73dv226d726gh23fnjncuyg0q9udfjf47eueu")
28            ]
29            cursor.executemany(
30                "INSERT INTO Usuarios (username, password, session_cookie)
31                VALUES (%s, %s, %s)",
32                usuarios_ejemplo
33            )
34            print("Usuarios de ejemplo insertados correctamente.")
35        else:
36            print("La tabla Usuarios ya contiene datos.")
37
38        cursor.close()
39        print("Tabla 'Usuarios' creada o verificada exitosamente en
40            PostgreSQL")
41        return True
42    except PBD.DatabaseError as error:
43        print("Error al crear la tabla o insertar usuarios en PostgreSQL")
44        print(error)
45        return False
```

## 6. Introducción al laboratorio

Se ha desarrollado un laboratorio interactivo que permite experimentar con diferentes tipos de inyecciones SQL en bases de datos Oracle y PostgreSQL. En los siguientes apartados se explicará en detalle cada inyección y cómo se ha implementado en el laboratorio.

## 7. Inyección basada en errores de la base de datos

La **inyección SQL basada en errores** es un tipo de ataque de inyección SQL en el que el atacante explota la información expuesta directamente por los mensajes de error generados por la base de datos. Estos mensajes de error proporcionan pistas sobre la estructura, el esquema y el contenido de la base de datos, lo que permite al atacante ejecutar consultas maliciosas para obtener acceso no autorizado a los datos o comprometer la integridad del sistema.

### 7.1. Mecanismo del Ataque

- **Explotación de Mensajes de Error:** Durante el desarrollo de aplicaciones web, los desarrolladores suelen habilitar mensajes de error detallados para depurar problemas. Si estos mensajes no se desactivan en producción, los atacantes pueden intencionalmente introducir consultas mal formadas o comandos SQL en los puntos de entrada de la aplicación (por ejemplo, formularios o parámetros URL) para provocar errores.
- **Uso de Consultas Maliciosas:** El atacante inserta código SQL diseñado para causar un error deliberado y obtener un mensaje detallado de la base de datos. Estos mensajes pueden revelar información sensible como nombres de tablas, nombres de columnas, tipos de datos o incluso fragmentos del contenido almacenado.
- **Iteración del Proceso:** Basándose en los datos recopilados de los mensajes de error, el atacante ajusta y refina sus consultas maliciosas para extraer información adicional o lograr un acceso más profundo al sistema.

Para acceder a la sección específica para esta inyección en el laboratorio, una vez desplegado el servidor con una apariencia como la siguiente:



Figura 2: Página de inicio del laboratorio de inyecciones SQL

Para acceder a la sección específica para esta inyección en el laboratorio, una vez desplegado el servidor, se debe seleccionar la opción *Database-Errors SQL Injection*.

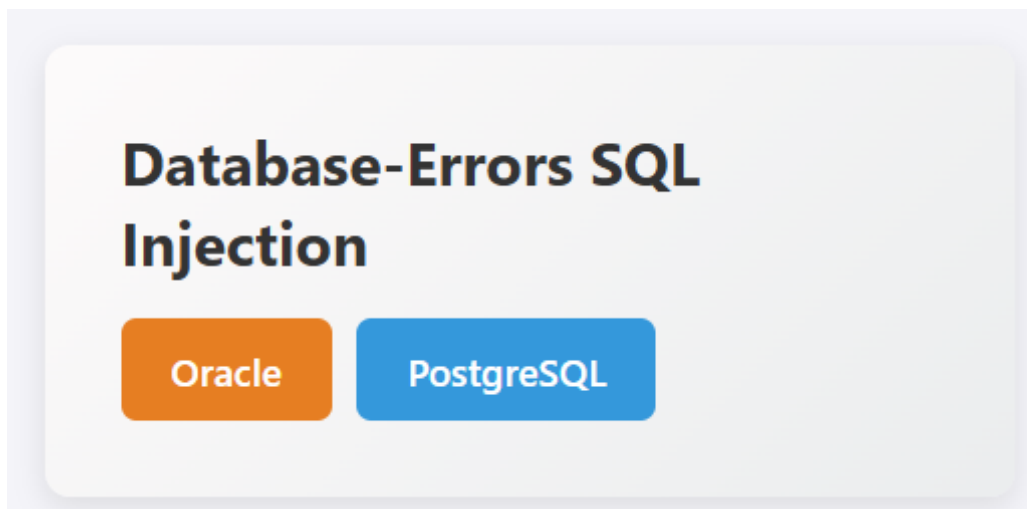


Figura 3: Opción de Database Errors en el laboratorio de inyecciones SQL

Como es posible observar, se tienen dos posibles opciones de trabajo de inyecciones en función de la base de datos que este trabajando de fondo, **Oracle** y **PostgreSQL**. Como va a ser común entre las diferentes secciones en el laboratorio, se tiene un formulario básico de inicio de sesión con dos campos: *usuario* y *contraseña*, simulando lo que podría ser una ventana de login en una página web genérica.

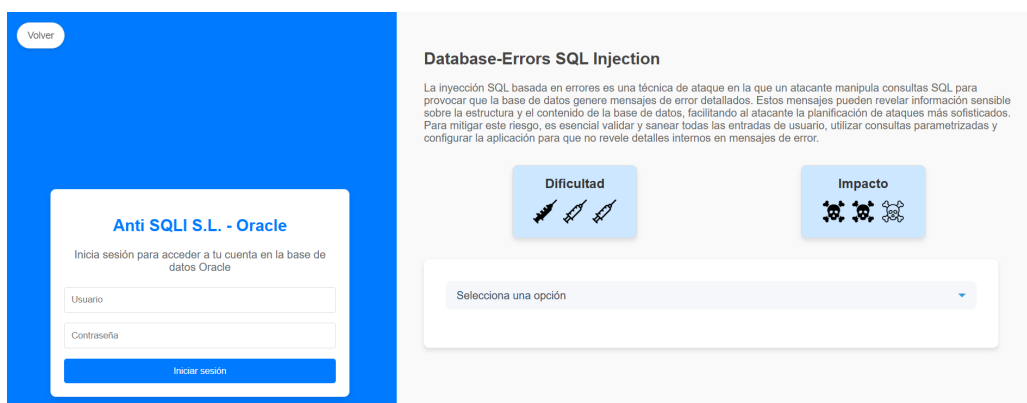


Figura 4: Formulario de inicio de sesión para inyección tipo Database-Errors

## 7.2. Login sin credenciales válidas

Como primer punto se ha decidido hacer mención al tipo de inyección para poder saltar el login sin credenciales válidas, ya que es un tipo de inyección fundamental y básica que va a funcionar en ambos SGBD y en todas las subsecciones del laboratorio.

De modo que para realizar dicha inyección en el selector de inyecciones de la derecha, debajo de las tarjetas de información relevante se deberá seleccionar el tipo de inyección "Login sin credenciales válidas" muestra la siguiente información para introducir en los campos **Usuario** y **Contraseña**.

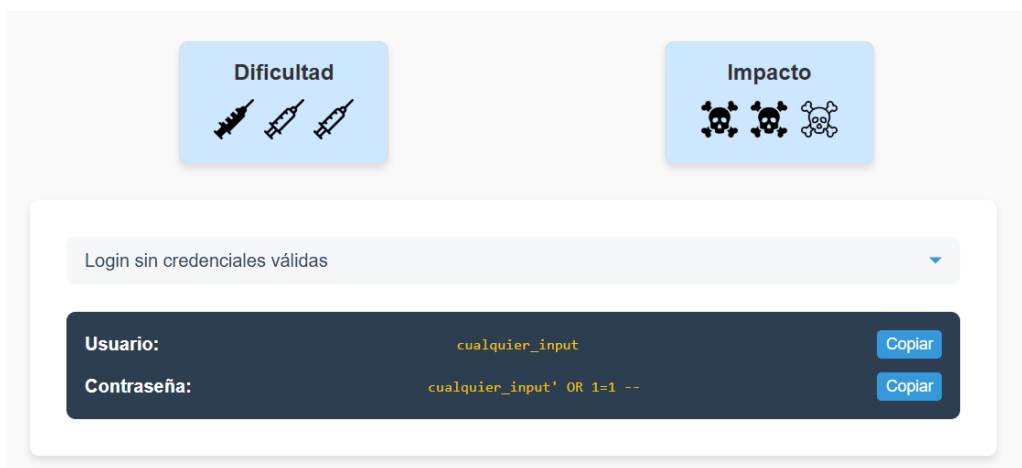


Figura 5: Datos a introducir en los campos del formulario

Como se puede observar en el código de inyección para el campo de la contraseña en el formulario, se aprovecha de la capacidad de SQL de poder trabajar con lógica booleana para inyectar que la comprobación de que el conjunto *usuario* y *contraseña* esten en una tupla de la tabla o que 1 es igual a 1 (comentando lo que venga detras para evitar comprobaciones secundarias), de modo que esta comprobación siempre va a ser cierta por la segunda parte de la comprobación.

```
1 SELECT * FROM Usuarios WHERE username = 'cualquier_input' AND password =
   'cualquier_input' OR 1=1
   -'
```

Atendiendo al diccionario de inyecciones en el código de la aplicación, en el caso de las inyecciones basadas en errores de la base de datos, se estaria usando la función con la vulnerabilidad que permita mostrar este tipo inyección, denominada "login\_inseguro\_errors\_oracle".

```
1 def login_inseguro_errors_oracle(username, password):
2     print("---login---")
3     print ("---login_inseguro_errors---")
4     conexion = dbConectarOracle() # Abre la conexión para autenticación
5     if not conexion:
6         print("Error: no se pudo conectar para autenticar.")
7         return False
8
9     sentencia = "SELECT * FROM Usuarios WHERE username = '"+username+"'
10    AND password = '"+password+"'"
11    try:
12        cursor = conexion.cursor()
13        cursor.execute(sentencia)
14        usuario = cursor.fetchone()
15
16        cursor.close()
17        dbDesconectar(conexion) # Cierra la conexión después de la
18        autenticación
19        if usuario:
20            print("Usuario autenticado:", usuario)
21            return {"resultado":usuario,"sentencia":sentencia, "auth":
22                    "true"}
23    else:
```

```

21         print("Usuario o contraseña incorrectos")
22         return {"sentencia":sentencia}
23     except PBD.DatabaseError as error:
24         print("Error al autenticar usuario")
25         print(error)
26         dbDesconectar(conexion)
27         return {"resultado":error, "sentencia":sentencia}

```

La vulnerabilidad se encuentra en la concatenación de los valores de los campos *usuario* y *contraseña* en la consulta SQL, lo que permite al atacante manipular la lógica de la consulta para obtener acceso no autorizado al sistema; tal y como se ha visto anteriormente.

Al ejecutar la función de inyección en el laboratorio, se obtiene un mensaje de éxito en la autenticación, lo que indica que la inyección ha sido exitosa y se ha logrado eludir la autenticación sin necesidad de credenciales válidas, pudiendo observar la sentencia SQL total ejecutada y la tupla resultado obtenida de la tabla, que debido a la naturaleza de la inyección, se ha devuelto la primera tupla de la tabla.

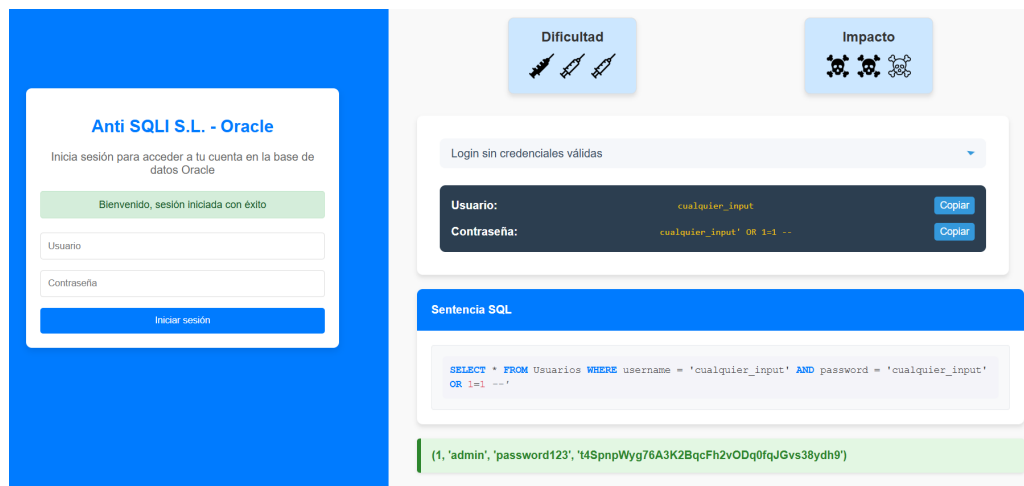


Figura 6: Resultado de la inyección para login sin credenciales válidas

### 7.3. información sobre las tablas

En la siguiente inyección se va a mostrar como se puede obtener información sobre las tablas de la base de datos, en este caso se va a obtener el nombre de las tablas de la base de datos. Para realizar dicha inyección en el selector de inyecciones de la derecha, debajo de las tarjetas de información relevante se deberá seleccionar el tipo de inyección **Información sobre las tablas** muestra la siguiente información para introducir en los campos **Usuario** y **Contraseña**.

### 7.4. Código vulnerable del login

## 8. Inyección basada en Union Attack

La inyección SQL basada en **UNION** es una técnica en la que un atacante utiliza la cláusula **UNION** para combinar los resultados de una consulta legítima con datos maliciosamente solicitados, permitiendo extraer información sensible de la base de datos. Para llevar a cabo este ataque, el atacante identifica puntos vulnerables en la aplicación web, determina el **número de columnas** en la consulta original y luego inyecta una consulta maliciosa que utiliza **UNION SELECT** para unir los resultados deseados. Para prevenir este tipo de ataques, es esencial



validar y sanear todas las entradas de usuario, utilizar consultas parametrizadas y aplicar el principio de privilegios mínimos en las cuentas de la base de datos.

## 8.1. Mecanismo del ataque

1. **Identificación de puntos vulnerables:** El atacante busca parámetros de entrada en la aplicación web que interactúan directamente con la base de datos. Esto puede incluir campos de formularios, parámetros en la URL, cookies o encabezados HTTP.
2. **Determinación del número de columnas:** Utilizando inyecciones como `ORDER BY` o consultas de prueba con `UNION SELECT NULL`, el atacante descubre el número de columnas en la consulta original. Esto asegura que la consulta maliciosa inyectada sea compatible con la estructura de la consulta legítima.
3. **Construcción de la inyección:** Una vez identificados los parámetros vulnerables y el número de columnas, el atacante construye una consulta maliciosa utilizando `UNION SELECT`. Por ejemplo: `http://example.com/page.php?id=1 UNION SELECT username, password FROM users`. En este caso, los datos sensibles de la tabla `users` son combinados con la consulta original.

Para acceder a la sección específica para esta inyección en el laboratorio, una vez desplegado el servidor, se debe seleccionar la opción *UNION-Attack SQL Injection*.



Figura 7: Opción de Union Attack en el laboratorio de inyecciones SQL

En esta sección, se tiene un formulario básico de inicio de sesión con dos campos: *usuario* y *contraseña*.



**Anti SQLi S.L. - Oracle**

Inicia sesión para acceder a tu cuenta en la base de datos Oracle

Usuario

Contraseña

**Iniciar sesión**

**UNION-Attack SQL Injection**

La inyección SQL basada en UNION es una técnica en la que un atacante utiliza la cláusula UNION para combinar los resultados de una consulta legítima con datos maliciosamente solicitados, permitiendo extraer información sensible de la base de datos. Para llevar a cabo este ataque, el atacante identifica puntos vulnerables en la aplicación web, determina el número de columnas en la consulta original y luego inyecta una consulta maliciosa que utiliza UNION SELECT para unir los resultados deseados. Para prevenir este tipo de ataques, es esencial validar y sanitizar todas las entradas de usuario, utilizar consultas parametrizadas y aplicar el principio de privilegios mínimos en las cuentas de la base de datos.

**Dificultad**

**Impacto**

Selecciona una opción

Figura 8: Formulario de inicio de sesión para Union Attack

## 8.2. Obtención de nombre de la base de datos

En primer lugar, una de las cosas más básicas que se pueden obtener con este tipo de inyección SQL es el nombre de la base de datos con la que se está operando. Esta información, aunque pueda parecer trivial, resulta fundamental para los atacantes, ya que les permite personalizar sus ataques dependiendo del sistema de gestión de bases de datos que esté en uso.

En este laboratorio, para realizar este ataque en Oracle, se utiliza una inyección SQL que combina la consulta legítima con un UNION SELECT. En concreto, el payload malicioso es el siguiente:

```
1 cualquier_input' UNION SELECT 1, ora_database_name, NULL AS
    nombre_bd_relleno_1, NULL AS nombre_bd_relleno_2 FROM dual --
```

En este caso:

- **ora\_database\_name**: Es una función específica de Oracle que devuelve el nombre de la base de datos activa.
- **NULL AS nombre\_bd\_relleno\_1** y **NULL AS nombre\_bd\_relleno\_2**: Se utilizan para completar el número de columnas requerido por la consulta original, ya que la cláusula UNION SELECT debe coincidir con el número y tipo de columnas de la consulta legítima.
- **dual**: Es una tabla especial de Oracle utilizada para ejecutar consultas que no necesitan datos de tablas reales.

### 8.2.1. Explicación del funcionamiento

La cláusula UNION SELECT combina los resultados de dos consultas SQL. En este caso, la inyección SQL modifica la consulta legítima original, añadiendo una nueva consulta que no está relacionada con los datos legítimos de la aplicación, pero que proporciona información sensible. Es importante tener en cuenta que la consulta maliciosa debe tener el mismo número de columnas que la consulta original. Por ejemplo, si la consulta legítima tiene cuatro columnas (en este caso en la tabla *usuarios* existen las columnas *id*, *username*, *password* y *session\_cookie*), la inyección debe proporcionar exactamente cuatro valores en su cláusula UNION SELECT. De lo contrario, Oracle devolverá un error debido a la incompatibilidad de columnas. Para esta inyección, solo interesa el valor de *ora\_database\_name*, pero se incluyen 1 y NULL como valores de relleno para las demás columnas requeridas.

Al ejecutar esta inyección, el atacante obtiene el nombre de la base de datos en el campo correspondiente, que se muestra en la interfaz del laboratorio como un valor destacado en la salida del ataque.

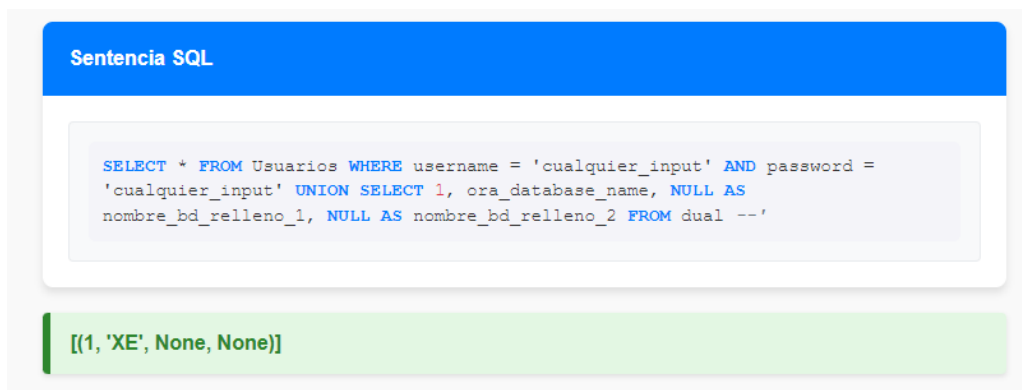


Figura 9: Obtención del nombre de la base de datos en Oracle

### 8.3. Obtención de la versión de la base de datos

Otro de los datos fundamentales que se pueden obtener mediante una inyección SQL basada en `UNION SELECT` es la versión de la base de datos en uso. Este dato proporciona información valiosa al atacante, ya que permite identificar la versión exacta del sistema de gestión de bases de datos (SGBD) Oracle. Con esta información, es posible adaptar los ataques a las vulnerabilidades específicas de esa versión.

En este laboratorio, se utiliza la siguiente inyección SQL para obtener la versión de la base de datos Oracle:

```
1 cualquier_input' UNION SELECT NULL, banner, NULL, NULL FROM v$version  
WHERE banner LIKE 'Oracle%' --
```

En este caso:

- **banner**: Es una columna de la vista `v$version`, que contiene información detallada sobre la versión y el entorno de la base de datos. Esta columna proporciona información como la edición de la base de datos, la versión y el sistema operativo sobre el que está ejecutándose.
- **v\$version**: Es una vista de diccionario del sistema en Oracle que almacena información sobre las versiones de los componentes del sistema de gestión de bases de datos. Esta vista es ampliamente utilizada tanto en administración legítima como en ataques para identificar detalles del entorno.
- **NULL**: Al igual que en otras inyecciones `UNION SELECT`, los valores `NULL` se utilizan para rellenar las columnas restantes en la consulta inyectada. Esto garantiza que la consulta inyectada tenga el mismo número de columnas que la consulta legítima original, evitando errores de sintaxis en la ejecución.
- **WHERE banner LIKE 'Oracle%'**: Este filtro se aplica para restringir los resultados de la consulta únicamente a las filas que contienen información relevante sobre la base de datos Oracle. La condición `LIKE 'Oracle%'` asegura que solo se devuelvan banners relacionados con la base de datos Oracle, excluyendo otros componentes potenciales del sistema.

### 8.3.1. Explicación del funcionamiento

El objetivo de esta inyección es aprovechar la estructura de la vista `v$version` para obtener la versión exacta de la base de datos. La consulta inyectada utiliza `UNION SELECT` para combinar los resultados de la consulta legítima con una consulta que devuelve el contenido de la columna `banner` de la vista `v$version`. Al ejecutar esta inyección, el valor de la versión se presenta en el campo correspondiente de la interfaz de la aplicación, proporcionando al atacante la información necesaria para ajustar ataques posteriores.

Al ejecutar esta inyección, el atacante obtiene la versión de la base de datos en el campo correspondiente, que se muestra en la interfaz del laboratorio como un valor destacado en la salida del ataque.

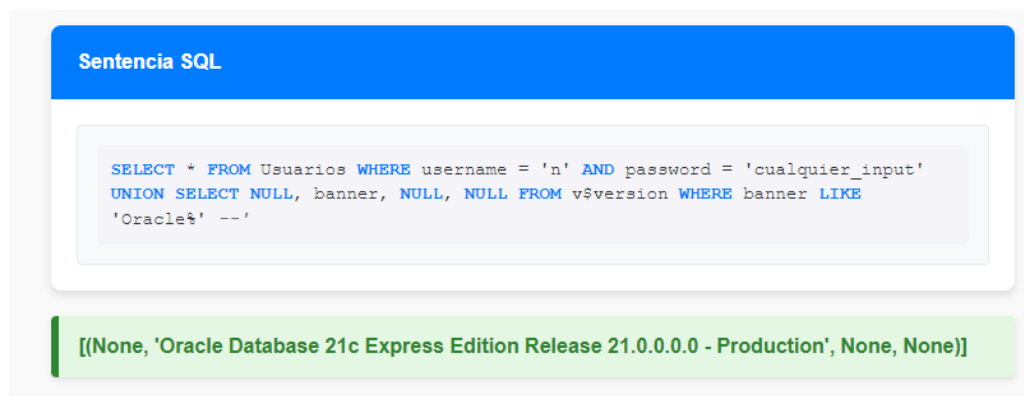


Figura 10: Obtención de la versión de la base de datos en Oracle

## 8.4. Obtención de todas las tablas de la base de datos

Un paso clave en un ataque avanzado es obtener un listado de todas las tablas de la base de datos. Este tipo de información permite a un atacante identificar las estructuras de datos disponibles, lo que facilita la selección de objetivos específicos, como tablas que almacenan credenciales, datos confidenciales o información sensible. Para lograr esto, se puede realizar una inyección SQL que extraiga información directamente de las vistas del sistema disponibles en la base de datos.

En este laboratorio, se utiliza la siguiente inyección SQL para obtener un listado de las tablas de la base de datos Oracle bajo el esquema del propietario `SYSTEM`:

```
1 cualquier_input' UNION SELECT 1, NULL, OWNER, TABLE_NAME FROM
  all_tables WHERE OWNER='SYSTEM' -- AND password = 'cualquier_input'
```

En este caso:

- **OWNER:** Es una columna de la vista `ALL_TABLES` que contiene el nombre del propietario (esquema) al que pertenece cada tabla en la base de datos. En este caso, se está filtrando específicamente al propietario `SYSTEM`, que normalmente contiene tablas relacionadas con la administración de la base de datos.
- **TABLE\_NAME:** Es otra columna de la vista `ALL_TABLES` que contiene el nombre de cada tabla dentro del esquema indicado. Esta es la información objetivo del ataque, ya que revela todos los nombres de las tablas disponibles.
- **NULL:** Se utiliza como valor de relleno para columnas que no son relevantes en la consulta inyectada. Esto asegura que el número de columnas en la consulta inyectada coincida con el de la consulta legítima, evitando errores de ejecución.

- **ALL\_TABLES**: Es una vista del diccionario de datos de Oracle que muestra todas las tablas accesibles al usuario actual, incluidas las de otros esquemas a las que tenga permisos.
- **WHERE OWNER='SYSTEM'**: Este filtro se aplica para limitar los resultados de la consulta a las tablas que pertenecen al esquema **SYSTEM**. Esto permite enfocar el ataque en un área específica de la base de datos.
- **- AND password = 'cualquier\_input'**: El uso del comentario (**-**) elimina cualquier condición adicional en la consulta original, como la verificación de contraseñas, asegurando que la consulta inyectada se ejecute sin restricciones. En este caso la inyección se realiza en el campo

#### 8.4.1. Explicación del funcionamiento

La consulta inyectada aprovecha la vista **ALL\_TABLES** para obtener un listado de todas las tablas accesibles en el esquema **SYSTEM**. La combinación de **UNION SELECT** con esta vista permite extraer datos estructurados directamente del diccionario de datos de Oracle.

La estructura de la consulta original se mantiene al incluir cuatro valores en la inyección (**1, NULL, OWNER y TABLE\_NAME**), lo que coincide con el número de columnas de la consulta legítima. Esto evita errores de sintaxis y asegura que los resultados de la inyección se combinen correctamente con los de la consulta original.

Al ejecutar esta inyección, el atacante obtiene todos los nombres de las tablas almacenadas en la base de datos.

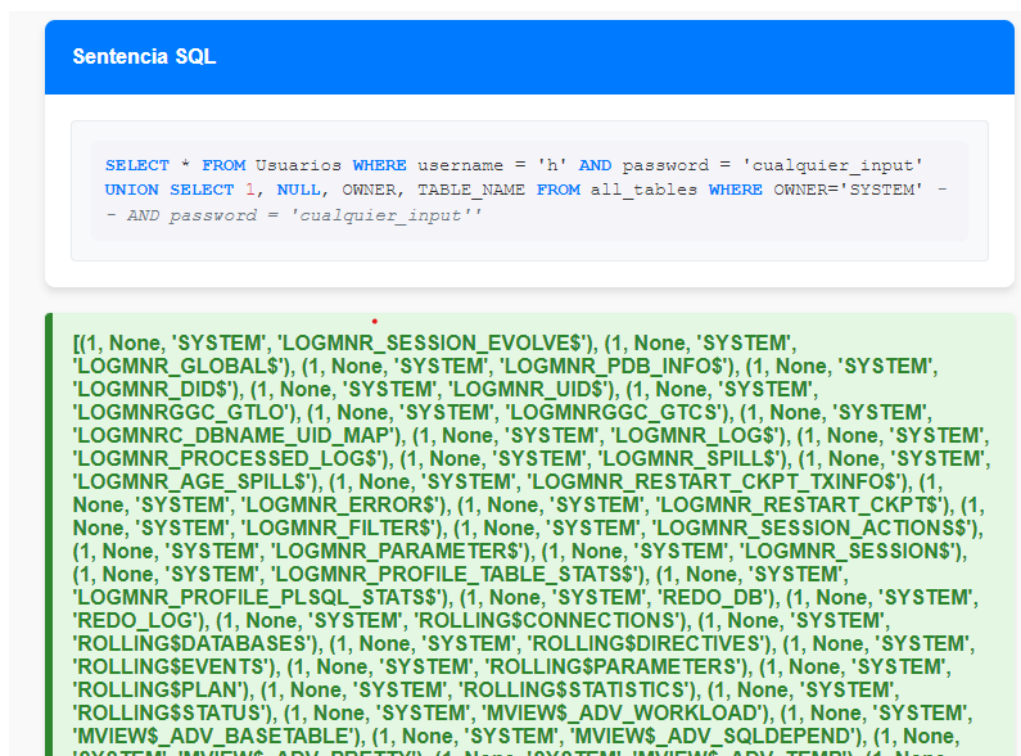


Figura 11: Obtención de tablas de la base de datos en Oracle

Al ejecutar una inyección SQL que consulta la vista **ALL\_TABLES**, es común que el resultado incluya tablas creadas por defecto por Oracle, como aquellas asociadas al sistema o utilizadas internamente para la administración de la base de datos. Para centrarse únicamente en tablas relevantes para el atacante, se pueden aplicar filtros adicionales en la cláusula **WHERE** de la inyección.

Un ejemplo de una inyección con estos filtros aplicados es el siguiente:

```
1 cualquier_input' UNION SELECT 1, NULL, OWNER, TABLE_NAME FROM
   all_tables
2 WHERE OWNER = 'SYSTEM'
3 AND TABLE_NAME NOT LIKE '%$%'
4 AND TABLE_NAME NOT LIKE 'SYS%'
5 AND TABLE_NAME NOT LIKE 'LOGMNR%' --
```

En este caso:

- `TABLE_NAME NOT LIKE '%$%'`: Filtra tablas cuyos nombres contienen el carácter \$, que generalmente son tablas de sistema utilizadas internamente por Oracle para la gestión de metadatos o componentes específicos.
- `TABLE_NAME NOT LIKE 'SYS%'`: Excluye tablas cuyos nombres comienzan con `SYS`, ya que estas suelen pertenecer al esquema del sistema (`SYS`) y no son de interés para la mayoría de los ataques.
- `TABLE_NAME NOT LIKE 'LOGMNR%'`: Elimina tablas relacionadas con la funcionalidad de LogMiner de Oracle, una herramienta utilizada para analizar registros de transacciones y que, en la mayoría de los casos, no contiene datos directamente útiles para los atacantes.

Al aplicar estos filtros, los resultados de la consulta se limitan a las tablas más relevantes, lo que facilita el análisis y reduce el ruido en el proceso de reconocimiento. Esto es especialmente útil en entornos con un gran número de tablas, donde los resultados pueden ser abrumadores si se incluyen todas las tablas creadas por defecto por Oracle.

Por ejemplo, al filtrar las tablas de sistema, el atacante puede enfocarse en tablas creadas por el administrador o los desarrolladores de la base de datos, que probablemente contengan información sensible o relacionada con las funcionalidades de la aplicación.

El laboratorio refleja esta estrategia de filtrado mostrando únicamente las tablas relevantes después de ejecutar la inyección filtrada. Esto permite a los participantes observar cómo una consulta más específica puede ser más efectiva en un ataque real.

**Sentencia SQL**

```
SELECT * FROM Usuarios WHERE username = 'k' AND password = 'cualquier_input'
UNION SELECT 1, NULL, OWNER, TABLE_NAME FROM all_tables WHERE owner = 'SYSTEM'
AND TABLE_NAME NOT LIKE '%$%' AND TABLE_NAME NOT LIKE 'SYS%' AND TABLE_NAME
NOT LIKE 'LOGMNR%' --'
```

```
[(1, None, 'SYSTEM', 'REDO_DB'), (1, None, 'SYSTEM', 'REDO_LOG'), (1, None, 'SYSTEM',
'SCHEDULER_PROGRAM_ARGS_TBL'), (1, None, 'SYSTEM', 'SCHEDULER_JOB_ARGS_TBL'),
(1, None, 'SYSTEM', 'REPL_VALID_COMPAT'), (1, None, 'SYSTEM',
'REPL_SUPPORT_MATRIX'), (1, None, 'SYSTEM', 'SQLPLUS_PRODUCT_PROFILE'), (1, None,
'SYSTEM', 'HELP'), (1, None, 'SYSTEM', 'USUARIOS)]]
```

Figura 12: Obtención de tablas filtradas en Oracle