



## **INFORME DE VULNERABILIDADES**

### **SQL INJECTION**

## 1.- INTRODUCCIÓN

Este informe analiza las vulnerabilidades de inyección SQL en dos scripts PHP de diferentes niveles de seguridad y proporciona recomendaciones para mejorar la seguridad.

## 2.- DESCRIPCIÓN

Un ataque de inyección SQL es una técnica que se utiliza para explotar las vulnerabilidades de seguridad en las aplicaciones web, consistente en insertar código SQL no autorizado, a través de campos de entrada vulnerables de una web, con el objetivo de extraer, modificar o borrar información sensible de la base de datos, ejecutar comandos en el sistema u obtener acceso a información confidencial.

Estos ataques son posibles cuando la aplicación no valida adecuadamente los datos de entrada proporcionados por el usuario y concatena directamente estos datos en una consulta SQL, permitiendo al atacante modificar la consulta SQL original.

## 3. ANALISIS Y EXPLOTACION DE VULNERABILIDADES

### 3.1.- BAJO NIVEL

```
<?php

if( isset( $_REQUEST[ 'Submit' ] ) ) {
    // Get input
    $id = $_REQUEST[ 'id' ];

    // Check database
    $query = "SELECT first_name, last_name FROM users WHERE user_id = '$id'";
    $result = mysqli_query($GLOBALS["__mysqli_ston"], $query ) or die( '<pre>' . ((is_object($GLOBALS["__mysqli_ston"]))
    ? mysqli_error($GLOBALS["__mysqli_ston"]) : (($__mysqli_res = mysqli_connect_error()) ? $__mysqli_res : false)) . '</pre>' );

    // Get results
    while( $row = mysqli_fetch_assoc( $result ) ) {
        // Get values
        $first = $row["first_name"];
        $last = $row["last_name"];

        // Feedback for end user
        echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
    }

    mysqli_close($GLOBALS["__mysqli_ston"]);
}

?>
```

Este código PHP ejecuta una consulta SQL en una Base de datos (BBDDs) para recuperar las credenciales de un usuario, desglosándose de la siguiente forma:

- 1- Primero, verifica si se ha enviado la solicitud comprobando si existe, siendo típico en formularios donde un botón de envío tiene el nombre *"Submit"*.
- 2- Si es cierto la opción 1, recupera el id del usuario desde la solicitud, siendo usado para buscar en la BBDDs.
- 3- Se produce la consulta a la BBDDs, construyendo una consulta SQL para seleccionar el nombre y apellido de un usuario basado en el id proporcionado. Aquí es donde el código es vulnerable a la inyección SQL (SQLi).
- 4- Se ejecuta la query en la BBDDs MySQL y maneja errores potenciales ("or die") dentro de etiquetas HTML *"<pre>"*. Si la query no es correcta devolverá *"false"*.
- 5- Si la consulta es correcta, recorre los resultados obtenidos en cada una de las filas, y muestra el id, el nombre y el apellido del usuario, usando nuevamente *"<pre>"* para facilitar la salida en HTML y cierra la conexión.

Teniendo en cuenta la descripción del código, para acceder a las credenciales de los 5 usuarios en la maquina DVWA, se ha procedido a modificar el campo *"user id"* de la query, debido a que todos los comandos SQL se encuentran concatenados, no existiendo separación entre ellos, existiendo una falta de sanitización en las entradas del usuario. Esta vulnerabilidad se ha explotado con una combinación de condiciones booleanas, siendo:



User ID:

```

ID: 'or '1'='1
First name: admin
Surname: admin

ID: 'or '1'='1
First name: Gordon
Surname: Brown

ID: 'or '1'='1
First name: Hack
Surname: Me

ID: 'or '1'='1
First name: Pablo
Surname: Picasso

ID: 'or '1'='1
First name: Bob
Surname: Smith

```

Realmente en el campo "User ID", solo se ha incluido la condición booleana, la cual iría acompañada de toda la query:

`$query = "SELECT first_name, last_name FROM users WHERE user_id = 'or '1'='1';( espacio, comilla, or, espacio, comilla,1, comilla, =, comilla,1)"`

En este caso, en la consulta original el '`$id`', se encuentra entrecomillado, y la inyección booleana realizada es incrustada, sustituyendo `$id`, por esta condición booleana, la cual, siempre es verdadera, por lo que nos aportará las credenciales de todos los usuarios.

Para mejorar este código y que deje de ser vulnerable a un ataque de inyección SQL, se podría utilizar la técnica de las sentencias preparadas:

```
<?php
// preguntamos si se ha enviado el formulario
if( isset( $_REQUEST[ 'Submit' ] ) ) {
    // obtenemos el id del usuario
    $id = $_REQUEST[ 'id' ];

    // preparamos la consulta para evitar inyecciones SQL (tecnica sentencias preparadas)
    $stmt = mysqli_prepare($GLOBALS["__mysqli_ston"], // preparamos la consulta

    // consulta SQL con un marcador de posición para el id()
    "SELECT first_name, last_name FROM users WHERE user_id = ?");

    // se vincula la preparación de la consulta con el marcador de posición de la id,
    // consiguiendo que el valor de la id y de la sentencia SQL no se mezclen, evitando inyecciones SQL
    mysqli_stmt_bind_param($stmt, "s", $id);

    // ejecutamos la consulta y guardamos el resultado en una variable
    mysqli_stmt_execute($stmt);
    $result = mysqli_stmt_get_result($stmt);

    // Si existen resultados, se mostraran al usuario
    while( $row = mysqli_fetch_assoc( $result ) ) {
        // Se obtiene los valores nombre y apellido del usuario
        $first = $row["first_name"];
        $last = $row["last_name"];

        // Se muestran los valores al usuario
        echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
    }

    // cierra la consulta y la conexión a la base de datos
    mysqli_stmt_close($stmt);
    mysqli_close($GLOBALS["__mysqli_ston"]);
}
?>
```

### 3.2.- NIVEL MEDIO

```
<?php

if( isset( $_POST[ 'Submit' ] ) ) {
    // Get input
    $id = $_POST[ 'id' ];

    $id = mysqli_real_escape_string($GLOBALS["__mysqli_ston"], $id);

    $query = "SELECT first_name, last_name FROM users WHERE user_id = $id;";
    $result = mysqli_query($GLOBALS["__mysqli_ston"],
    $query) or die( '<pre>' . mysqli_error($GLOBALS["__mysqli_ston"]) . '</pre>' );

    // Get results
    while( $row = mysqli_fetch_assoc( $result ) ) {
        // Display values
        $first = $row["first_name"];
        $last = $row["last_name"];

        // Feedback for end user
        echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
    }
}

// This is used Later on in the index.php page
// Setting it here so we can close the database connection in here like in the rest of the
$query = "SELECT COUNT(*) FROM users;";
$result = mysqli_query($GLOBALS["__mysqli_ston"],
$query ) or die( '<pre>' . ((is_object($GLOBALS["__mysqli_ston"]))
    ? mysqli_error($GLOBALS["__mysqli_ston"]) : ($___mysqli_res = mysqli_connect_error())
    || $___mysqli_res : false));
$number_of_rows = mysqli_fetch_row( $result )[0];

mysqli_close($GLOBALS["__mysqli_ston"]);
```

Como el código anterior, igualmente, realiza una consulta a una BBDDs para la obtención de credenciales de usuarios basados en un id proporcionado en la query, de la siguiente forma:

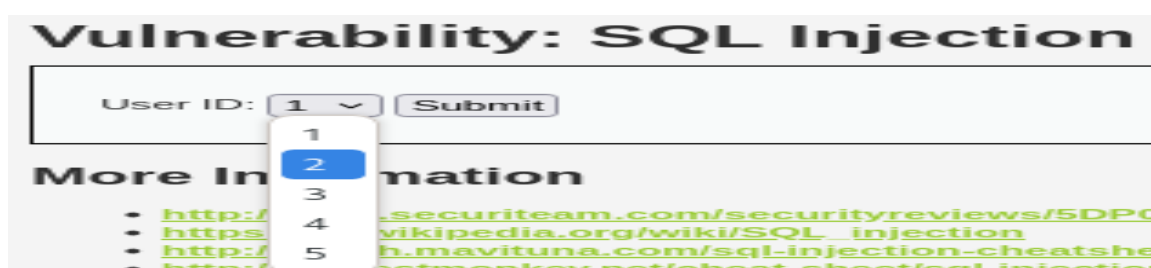
- 1- Los dos primeros pasos coinciden con el código analizado de nivel bajo: Se verifica si el botón “Submit” se ha presionado y ha enviado el formulario, para obtener una “id” de usuario conforme a dicha petición.
- 2- Se ponen medidas para escapar de caracteres especiales en la cadena “id” en la realización de consultas y evitar ataques SQLi, como son: NUL (carácter nulo, \0), Nueva línea (\n), Retorno de carro (\r), Comilla simple ('), Comilla doble (") ,el Carácter de escape (\) y Control-Z (\x1A).
- 3- Como anteriormente. Se prepara la consulta SQL para ejecutarla en la BBDDs, siendo el código es vulnerable a inyecciones SQL. En este caso, el marcador \$id esta sin comillas simples, por lo que para intentar el ataque SQLi se usara la condición booleana: 1 or 1=1 --.



Esto quiere decir que la consulta se convierte en 1, que selecciona todos los usuarios, ya que 1=1 siempre es verdadero y con este carácter – o este #, se comenta el resto de la consulta, dejándola sin efecto alguno.

- 4- El resto de código, coincide con los puntos 4 y 5 del análisis realizado al código de bajo nivel, es decir, consulta y manejo de errores de MySQL, obtención y muestra de resultados al usuario.

Después de este pequeño análisis, observamos en la web de DVWA que no existe un campo para poder escribir la inyección SQL, sino un desplegable con 5 ID de los usuarios:



Así que, se procede a capturar / interceptar el tráfico de la web DVWA con la aplicación BurpSuite, donde se cambiará el “id” por la condición booleana, llevando a cabo el ataque SQLi.

Captura original BurpSuite

```
POST /vulnerabilities/sqli/ HTTP/1.1
Host: 10.0.2.6
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/png,image/svg+xml,*/*;q=0.8
Accept-Language: es-ES,es;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded
Content-Length: 18
Origin: http://10.0.2.6
Connection: keep-alive
Referer: http://10.0.2.6/vulnerabilities/sqli/
Cookie: PHPSESSID=670b3687aneeilnctbaqpdvmc9; security=medium
Upgrade-Insecure-Requests: 1
Priority: u=0, i

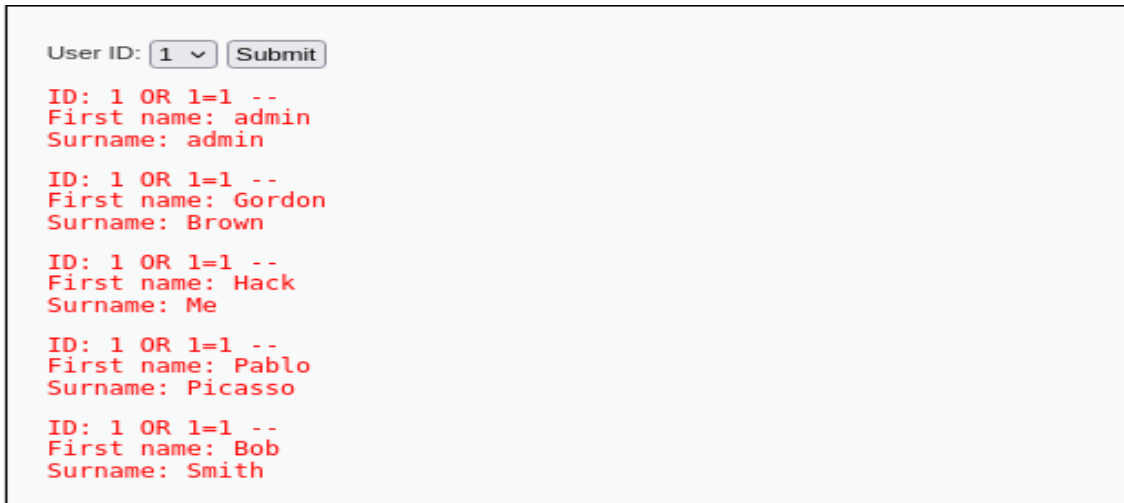
id=1&Submit=Submit
```

captura BurpSuite modificada SQLi

```
POST /vulnerabilities/sqli/ HTTP/1.1
Host: 10.0.2.6
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/png,image/svg+xml,*/*;q=0.8
Accept-Language: es-ES,es;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded
Content-Length: 18
Origin: http://10.0.2.6
Connection: keep-alive
Referer: http://10.0.2.6/vulnerabilities/sqli/
Cookie: PHPSESSID=670b3687aneeilnctbaqpdvmc9; security=medium
Upgrade-Insecure-Requests: 1
Priority: u=0, i

id=1 OR 1=1 -- &Submit=Submit
```

Una vez modificada, se envía de nuevo al servidor “DVWA”, pulsando la opción “*Foward*” de BurpSuite, obteniendo como resultado las credenciales de los 5 usuarios:



```

User ID: 1 Submit

ID: 1 OR 1=1 --
First name: admin
Surname: admin

ID: 1 OR 1=1 --
First name: Gordon
Surname: Brown

ID: 1 OR 1=1 --
First name: Hack
Surname: Me

ID: 1 OR 1=1 --
First name: Pablo
Surname: Picasso

ID: 1 OR 1=1 --
First name: Bob
Surname: Smith
  
```

Para prevenir inyecciones SQL, se deben utilizar sentencias preparadas con parámetros vinculados, como se muestra en el siguiente código:

```

<?php
// Se Comprueba si se ha enviado el formulario y se obtiene el id del usuario
if( isset( $_POST[ 'Submit' ] ) ) {
    $id = $_POST[ 'id' ];

    // Se prepara la consulta a La BBDDs "GLOBALS['__mysqli_ston']" de MySQL, usando la tecnica de sentencias preparadas,
    //usando en user_id un marcador de posición (?), vinculando el valor de la id con la consulta, evitando inyecciones SQL
    $stmt = mysqli_prepare($GLOBALS['__mysqli_ston'], "SELECT first_name, last_name FROM users WHERE user_id = ?");
    mysqli_stmt_bind_param($stmt, "s", $id);

    // Se ejecuta la consulta y se guarda el resultado en una variable para mayor seguridad
    mysqli_stmt_execute($stmt);
    $result = mysqli_stmt_get_result($stmt);

    // Se obtienen los valores de nombre y apellido
    // y se muestra el resultado al usuario en formato html
    while( $row = mysqli_fetch_assoc( $result ) ) {
        $first = $row["first_name"];
        $last = $row["last_name"];
        echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
    }

    // Se finaliza la sentencia y la conexión
    mysqli_stmt_close($stmt);
}
mysqli_close($GLOBALS['__mysqli_ston']);
?>
  
```

## 4.- RECOMENDACIONES DE SEGURIDAD

1. Las consultas preparadas son la mejor defensa contra la inyección SQL, ya que separan la lógica de la consulta de los datos, evitando que los datos maliciosos alteren la consulta, como se ha podido ver en los archivos PHP de mejora de código planteados anteriormente.
2. Validar y sanitizar, siempre, la entrada del usuario, para asegurar que los datos sean del tipo y formato esperados.
3. Como en los ataques LFI y RFI, establecer el principio de privilegios mínimos de la cuenta de la base de datos utilizada por la aplicación.
4. Implementar medidas de registro y monitoreo, para detectar y responder a intentos de inyección SQL.
5. El código usado en el nivel medio es vulnerable a inyecciones SQL debido a la inclusión directa del id en la consulta SQL, sin una validación o saneamiento adecuado, ya que, aunque se utiliza “*mysqli\_real\_escape\_string*” para escapar de ciertos caracteres especiales que se han especificado anteriormente, no es suficiente para prevenir inyecciones SQL en todos los casos.
6. Es importante señalar que, tanto en el código PHP de nivel bajo y medio usan un método de manejo de errores útil durante el desarrollo, pero no es recomendable para un entorno de producción debido a que puede revelar detalles sensibles sobre la base de datos o la lógica del programa.

```
// Check database
$query = "SELECT first_name, last_name FROM users WHERE user_id = '$id'";
$result = mysqli_query($GLOBALS["__mysqli_ston"], $query ) or die( '<pre>' . ((is_object($GLOBALS["__mysqli_ston"]))
? mysqli_error($GLOBALS["__mysqli_ston"]) : (($__mysqli_res = mysqli_connect_error()) ? $__mysqli_res : false)) . '</pre>' );
```



En un entorno de producción, sería mejor manejar los errores de manera que no exponga información sensible y que proporcione una respuesta adecuada al usuario, como podría ser:

```
<?php
....
....
....
// Se preparara la consulta para evitar inyecciones SQL, usando la técnica de sentencias preparadas
$stmt = mysqli_prepare($GLOBALS["__mysqli_ston"], "SELECT first_name, last_name FROM users WHERE user_id = ?");
if (!$stmt) {
    // Antes de la vinculacion del marcador, se comprueba si el servidor de BBDDs puede preparar la consulta,
    // En caso negativo, se muestra un mensaje de error
    error_log("Error de preparación de consulta: " . mysqli_error($GLOBALS["__mysqli_ston"]));
    // Se muestra un mensaje genérico al usuario
    die("Ocurrió un error. Por favor, inténtelo de nuevo más tarde.");
}
// Se vincula el valor de la id con la sentencia SQL preparada
mysqli_stmt_bind_param($stmt, "s", $id);

// Se ejecuta la consulta y se guardan el resultado en una variable
mysqli_stmt_execute($stmt);
$result = mysqli_stmt_get_result($stmt);

// Se comprueba si la consulta se ha podido ejecutar
if ($result === false) {
    // si la consulta no se puede ejecutar, se muestra un mensaje de error
    error_log("Error al ejecutar la consulta: " . mysqli_error($GLOBALS["__mysqli_ston"]));
    // Mostrar un mensaje genérico al usuario
    die("Ocurrió un error. Por favor, inténtelo de nuevo más tarde.");
}
....
....
....
?>
```

Este enfoque mejora significativamente la seguridad y la robustez del código al prevenir inyecciones SQL y manejar los errores de manera adecuada en un entorno de producción.

## 5. CONCLUSIONES

Las vulnerabilidades por inyección SQL, representan evidentes riesgos para la seguridad de las aplicaciones web, siendo muy importante para las organizaciones implementar prácticas de codificación segura, validación y sanitización todas las entradas del usuario, configurar adecuadamente el servidor para mitigar estos riesgos, así como el empleo de técnicas avanzadas como ML y DL para automatizar la detección y prevención de estos ataques de manera efectiva, proporcionando una capa adicional de seguridad para identificar patrones anómalos en las consultas SQL.

Durante este informe, se ha analizado cómo las consultas SQL, pueden ser vulnerables a ataques de inyección SQL (SQLi) cuando no se escapan adecuadamente las entradas del usuario, con el uso de expresiones condicionales booleanas, como `'or '1'='1' o 1 OR 1=1 --`, consiguiendo truncar las consultas SQL, haciendo que las mismas sean verdaderas, devolviendo todos los registros. Pero aún, se pueden usar muchas más expresiones como pueden ser:

- Combinación de consultas y resultados:  
`" UNION SELECT username, password FROM users --"`
- Uso de subconsultas para extracción de datos:  
`' OR (SELECT COUNT(*) FROM users) > 0 --`

Para prevenir estos ataques utilizar las recomendaciones planteadas anteriormente, destacando la importancia de la validación y saneamiento de las entradas del usuario y una codificación correcta en la fase de desarrollo de aplicaciones web.

Para terminar, y teniendo en cuenta el desarrollo de la Inteligencia Artificial, el uso de técnicas de Machine Learning (ML) o Deep Learning (DL), pueden ayudar a detectar patrones de comportamiento anómalos y prevenir ataques de inyección SQL, con modelos entrenados en identificar solicitudes sospechosas basadas en características de las consultas SQL, como, por ejemplo:

- En primer lugar, el proceso de recolección de datos de tráfico web, incluyendo consultas SQL legítimas y maliciosas, guardándola en formato .CSV.
- En segundo lugar, se hará un preprocesamiento de datos, para convertir las consultas SQL en vectores numéricos con características, para que pueda ser entrenado el modelo.
- Para iniciar el entrenamiento, se pueden usar algoritmos de ML como *Random Forest*, *decisión tree*, *GradientBosster*, *KNN*, etc o redes neuronales mediante el uso del *Transfer Learning* o el *Fine Tunning* para entrenar un modelo de detección de SQLi, aprovechando modelos entrenados como *ResNet*, *AlexNet*, *DenseNet*, etc, para conseguir un mayor poder computacional con recursos locales limitados.
- Ejemplo realizado con ML:

```
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

# Recolección de datos (simulada)
data = pd.read_csv('sql_queries.csv')

# Preprocesamiento de datos con conteo de palabras (bag of words)
# para lograr representación numérica de las palabras en las consultas
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(data['query'])
y = data['label']

# División de datos en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Entrenamiento del modelo
model = RandomForestClassifier()
model.fit(X_train, y_train)

# Evaluación del modelo
y_pred = model.predict(X_test)
print(classification_report(y_test, y_pred))
```

Una vez entrenado el modelo y obtenidos buenos resultados en las predicciones, se podría implementar en una función para la detección de SQLi:

```
# Función para detección de SQL Injection con Machine Learning
def detection_SQLi(query):
    # Se convierte la consulta a un vector numérico
    query_vector = vectorizer.transform([query])
    # Se realiza la predicción con el modelo anteriormente entrenado
    prediccion = model_SQLi.predict(query_vector)
    # Se retorna si la consulta es maliciosa o no,
    # suponiendo que las etiquetas son 'malicious' y 'safe'
    return prediccion[0] == 'malicious'
```

```
# Ejemplo de uso
query = "SELECT * FROM users WHERE id = '1 OR 1=1 --'"
if detection_SQLi(query):
    print("SQL Injection detected!")
else:
    print("Query is safe.")
```

Para finalizar, significar que la implementación de las recomendaciones anteriores mejorará significativamente la seguridad de las aplicaciones web y protegerá contra ataques de inyección SQL.