TRABAJO DE FIN DE MÁSTER

Seguridad en aplicaciones Web

Máster interuniversitario en seguridad de las tecnologías de la información y de las comunicaciones (MISTIC)
Área: Seguridad en servicios y aplicaciones
Autor: Diego Losada Regos
Tutores: Jordi Duch Gavaldà y Agustí Solanas Gómez
Junio de 2015

Resumen

En el presente trabajo se ha implementado una aplicación web que permitirá a un usuario no experto realizar un análisis de una determinada vulnerabilidad web en un servidor externo a la aplicación. Esos análisis se almacenarán en una base de datos y se podrán consultar en cualquier momento por dicho usuario.

La vulnerabilidad web escogida para el desarrollo de la aplicación ha sido SQL *Injection*. Dicha vulnerabilidad ha sido catalogada como la más crítica por OWASP por lo que cobra especial interés desarrollar una aplicación capaz de realizar un escáner completo de dicha vulnerabilidad en un servidor externo. Ante la complejidad de desarrollar un escáner propio he decidido integrar la herramienta *sqlmap* en la aplicación, desde donde se realizarán los diferentes análisis.

Para facilitar la navegación a lo largo de la aplicación he decidido desarrollar una web dinámica, en donde destacan entre otros el uso del *framework* jQuery en la parte *front-end* y del *framework* Symfony en el *back-end*.

Para probar el correcto funcionamiento de la aplicación se ha desarrollado una Web de pruebas externa. Tanto la web principal como la web de pruebas son accesibles desde Internet en donde se puede probar el producto desarrollado. Las URLs correspondientes son: http://tfmdlrprod.es/ y http://tfmdlrtest.es/

Abstract

In this final work I have developed a web application from which a non-expert user will be able to make a scanner of a chosen web vulnerability on a external server. The results will be stored in a data base and the user will be able to check them whenever he or she wants.

The chosen web vulnerability was SQL Injection. That vulnerability has been cataloged as the most dangerous vulnerability by OWASP. For that reason, It is interesting to make a web application which helps a non experts users to avoid the consequences. To develop a full SQL Injection scanner it is a huge work and it needs a lot of time, so I have decided to integrate the sqlmap tool as part of the application.

To make the navigation along the application easier I have decided to make a dynamic web application. The most important tool in the front-end will be the framework jQuery, and in the back-end will be the framework Symfony.

To test the final application I have developed an external Web. Both applications are accesible from Internet: http://tfmdlrprod.es/ and http://tfmdlrtest.es/

Palabras clave

SQL Injection, sqlmap, Symfony, jQuery, escáner, vulnerabilidad web.

Índice

Tabla de imágenes

Tabla de ilustraciones

1. Introducción

1.1. Justificación del TFM y contexto

Mi decisión al escoger este TFM se basó en que me permitía ampliar mis conocimientos en desarrollo de aplicaciones web. Mi experiencia previa se basaba en entornos con Node.js y PHP plano, pero no tenía ninguna experiencia con el *framework* Symfony. Este proyecto me ha brindado la oportunidad de ampliar mis conocimientos de programación en PHP, sobretodo en dicho *framework*.

Con respecto a la elección de SQL *Injection* como vulnerabilidad a tratar me he basado en que es una de las más críticas y una de las más extendidas. Poder realizar un análisis de SQL *Injection* desde un navegador web me ha parecido una idea que puede ahorrar trabajo a la hora de realizar pruebas de penetración, sobre todo en desarrolladores con poca experiencia.

Por último, la elección de sqlmap como herramienta para realizar los análisis ha sido debida a mi experiencia con dicha herramienta a lo largo del MISTIC. La he utilizado en las prácticas de las asignaturas “Vulnerabilidades de seguridad” y “Seguridad en bases de datos”. Me ha parecido una herramienta cómoda y con una lista de comandos extensa que me podría facilitar el trabajo a la hora de integrarla en la aplicación.

1.2. Objetivos del TFM

Objetivo Principal. Implementar una aplicación Web capaz de realizar análisis de vulnerabilidades SQL *Injection* en servidores externos, procesar la información obtenida y mostrarla al usuario. La aplicación deberá almacenar los resultados para mostrarlos al usuario cuando él quiera. Se debe desarrollar la aplicación para que pueda ser utilizada por usuarios no expertos en seguridad informática.

Para alcanzar el objetivo principal será necesario alcanzar la siguiente lista de objetivos:

Objetivo 1. Implementar un sistema seguro de alta y autentificación de los usuarios. Cada usuario tendrá acceso a sus análisis realizados y sólo a los suyos por lo que es importante implementar un sistema cerrado, que no permita a los usuarios consultar los análisis realizados por otros.

Objetivo 2. Crear una interfaz dinámica desde la cual el usuario pueda realizar un análisis de vulnerabilidades SQL *Injection* en un servidor externo. Se compondrá de una lista de parámetros que el usuario tendrá que rellenar para poder realizar dicho análisis.

Objetivo 3. Estudiar la forma de integrar la herramienta sqlmap dentro de una aplicación en PHP. Una vez integrada crear una interfaz lógica que nos permita realizar los análisis dependiendo del tipo de parámetros enviados.

Objetivo 4. Recoger y hacer un tratamiento de la información resultante de sqlmap.

Objetivo 5. Almacenar los resultados del análisis en una base de datos para su posterior visualización.

Objetivo 6. Implementar un apartado en donde el usuario pueda navegar entre los resultados de los distintos servidores.

Objetivo 7. Crear una página en donde el usuario pueda consultar cómo funciona el sistema y el tipo de parámetros que debe introducir.

Objetivo 8. Crear una web diferente, vulnerable a ataques SQL *Injection* a través de la cual podamos comprobar el funcionamiento de la web principal.

1.3. Herramientas de desarrollo

Dentro de la aplicación se podrían definir dos particiones lógicas: el *front-end* y el *back-end*.

A grandes rasgos, el *front-end* es el conjunto de tecnologías que están en contacto con el usuario (visualización de contenido, recogida y envío de datos, etc.). Por su parte, el *back-end* es el conjunto de tecnologías que están en contacto con el servidor (tratamiento de peticiones, conexión con la base de datos, etc.). Las tecnologías utilizadas en este proyecto en ambas partes son las siguientes:

- *Front-end*: HTML, CSS y JavaScript. Para facilitar el trabajo con CSS he utilizado el *framework* Bootstrap [10] y con JavaScript he utilizado jQuery [9].
- *Back*-end: PHP, con el *framework* Symfony sobre un servidor Apache. La base de datos escogida ha sido MySQL.

1.4. Planificación del proyecto

La realización de este proyecto se llevará a cabo en los siguientes pasos:

1.4.1. Estudio de la guía OWASP

El primer paso será realizar un estudio de la guía de vulnerabilidades OWASP [1]. Una vez elegida la vulnerabilidad, se procederá a un estudio de las herramientas con las que poder escanear servidores externos.

1.4.2. Estudio de sqlmap

Estudio de la herramienta sqlmap [2], su forma de funcionar y las posibilidades que ofrece.

1.4.3. Estudiar y comprender Symfony

Symfony [8] es completamente nuevo para mí por lo que necesito un cierto tiempo para comprender la estructura y el funcionamiento de una aplicación desarrollada con dicho *framework*.

1.4.4. Diseño e implementación página Login

La página de Login es la que da entrada al sistema y donde se efectúa el alta y la autentificación del usuario. Es por ello que en este apartado se necesita la base de datos para comprobar o insertar usuarios. El sistema de autentificación será el clásico de usuario/contraseña en un paso.

1.4.5. Diseño e implementación página principal

La página principal es la encargada de recoger los datos necesarios para efectuar los análisis en servidores externos, enviarlos al servidor y realizar el análisis. Por tanto, es donde se realiza la integración de sqlmap a la aplicación.

1.4.6. Diseño e implementación página historial

La página de historial es desde donde el usuario será capaz de visualizar los resultados de los diferentes análisis de los diferentes servidores.

1.4.7. Diseño e implementación página información

La página de información será la más sencilla del sistema. Su función será indicar al usuario cómo funciona la aplicación y qué debe introducir en cada parámetro para obtener los resultados deseados en la aplicación.

1.4.8. Desarrollo de web de pruebas

La web de pruebas será una web externa a la aplicación diseñada para ser vulnerable a ataques SQL *Injection*. Nos servirá para comprobar el perfecto funcionamiento de la aplicación principal. Su desarrollo ha sido en paralelo a la aplicación principal debido a la necesidad de probar el código desarrollado en cada momento.

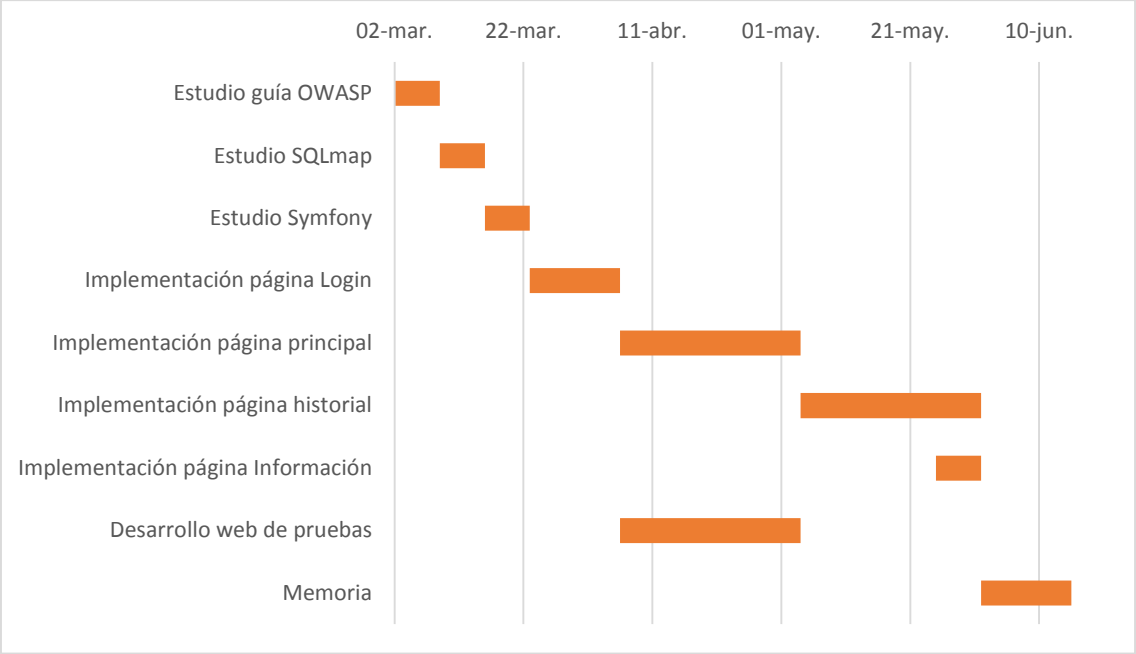

Ilustración 1 - Diagrama de Gantt del TFM

1.5. Descripción de capítulos de la memoria

Para entender mejor los diferentes capítulos de la memoria voy a realizar una pequeña descripción de cada uno de ellos:

Capítulo 1 - Introducción: Capítulo de entrada de la memoria que sirve para entender el proyecto.

Capítulo 2 – SQL Injection y sqlmap: Capítulo en donde explicamos la vulnerabilidad a escanear en la aplicación y la herramienta empleada.

Capítulo 3 – Estructura lógica de la aplicación: Capítulo en donde explicamos la estructura lógica de la aplicación. Módulos principales y relación entre ellos.

Capítulo 4 – Symfony y estructura del código: Capítulo que permite entender el entorno de desarrollo del *back-end* que servirá como base para comprender la estructura del código del proyecto y la comunicación entre las diferentes partes.

Capítulo 5 - Diseño e implementación *front-end*: En este capítulo se exponen y justifican las decisiones tomadas en la parte *front-end*, así como explicar las funcionalidades de cada parte.

Capítulo 6 - Implementación *back-end*: Este capítulo es el equivalente al capítulo anterior pero centrado en el *back-end.*

Capítulo 7 - Integración sqlmap: Se expone la forma en la que se ha integrado sqlmap a la aplicación y cómo se ha utilizado para realizar los diferentes análisis.

Capítulo 8 – Web de pruebas y ejemplo de uso de la aplicación principal: Se explica en qué consiste la web de pruebas y por qué sirve para comprobar el funcionamiento de la aplicación principal. Ilustraremos el funcionamiento de ambas aplicaciones con un ejemplo.

Capítulo 9 – Conclusiones y líneas futuras: Conclusiones del TFM y líneas futuras.

Capítulo 10 – Bibliografía: Fuentes consultadas para la elaboración del trabajo.

2. SQL *Injection* y sqlmap

En este capítulo vamos a explicar en profundidad la problemática de SQL *Injection*. Posteriormente haremos una pequeña introducción a la herramienta sqlmap.

2.1. SQL Injection

¿Qué es?

Antes de definir el ataque SQL *Injection* es importante saber sus orígenes, que no son otros que SQL. SQL es un lenguaje declarativo de acceso a bases de datos que permite consultar, insertar y modificar la información. Su sencillez y longevidad hicieron que SQL sea el lenguaje de acceso de base de datos más utilizado a día de hoy. Gestores de bases de datos como MySQL, Oracle o SQLite entre muchos otros están basados en SQL. Esto ha provocado que la gran mayoría de lenguajes de programación hayan implementado librerías para trabajar con SQL, lo que ha derivado en su utilización masiva en cualquier tipo de aplicaciones.

El contexto para que se produzca un ataque SQL *Injection* es el siguiente:

- Aplicación que consulta una base de datos mediante el lenguaje SQL.
- La aplicación recibe datos de una fuente desconocida.
- La aplicación ejecuta consultas a la base de datos de forma dinámica.

El ataque SQL *Injection* consiste en enviar código SQL a la aplicación, y si la aplicación es vulnerable, ejecutará el código SQL enviado por el atacante, dándole la posibilidad de obtener o modificar la información almacenada. Este ataque es verdaderamente peligroso, hasta el punto de que la guía OWASP ha declarado el ataque SQL *Injection* como el más crítico en un sistema [5].

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific

Imagen 1 – Puntuación de SQL Injection, por OWASP

Como podemos observar, destaca en su sencillez y su impacto, lo que deriva en consecuencias devastadoras para el sistema. Esto sumado al gran número de aplicaciones que utilizan SQL hace que este tipo de ataque sea el más extendido. Si un atacante es capaz de inyectar su propio código en una consulta puede obtener o modificar cualquier información de la base de datos (contraseñas, estructura, privilegios, etc.).

Consecuencias de ser vulnerable a SQL Injection

- **Descubrimiento y modificación de información:** A través de la modificación de las consultas a una base de datos el atacante puede obtener y modificar casi cualquier información almacenada. Esto afecta a las propiedades de seguridad de confidencialidad, integridad y no repudio.
- **Elevación de privilegios:** En caso de que se almacenen contraseñas en una base de datos vulnerable puede permitir al atacante acceder al sistema con la cuenta de usuarios privilegiados. Esto afecta a la propiedad de autorización.
- **Denegación de servicio:** La inyección de código no solo se puede utilizar para obtener información sino también para borrarla o modificarla. El efecto

más probable a este ataque es el colapso del sistema. Afecta a la propiedad de disponibilidad.

- **Suplantación de usuarios:** La obtención de credenciales permitirá a un atacante acceder al sistema haciéndose pasar por una persona autorizada. Afecta a la propiedad de autentificación.

En definitiva, una vulnerabilidad SQL *Injection* puede llegar a violar todas las propiedades de seguridad de un sistema informático.

¿Cómo se produce?

Un ataque de inyección de SQL viene precedido por una mala gestión de los datos recibidos para la consulta. El origen de este ataque reside en la capacidad del sistema en interpretar los datos recibidos como código ejecutable. Imaginemos que un sistema de autentificación en PHP que utiliza una base de datos MySQL. Para acceder, el usuario envía su alias y su contraseña. La aplicación recibe ambos parámetros y ejecuta la siguiente consulta SQL:

```
SELECT count(*) FROM usuarios WHERE usuario='$usuario' AND password='$password';
```

Donde '$usuario' y '$password' son datos enviados por un usuario. Solamente accederá al sistema si la consulta anterior devuelve un número mayor de cero (la consulta cuenta todas las filas que cumplen la condición del 'WHERE' en la tabla usuarios). Imaginemos que un usuario malintencionado envía $usuario=Cualquiera y $password= ' OR '1'='1. La consulta quedaría de la siguiente forma:

```
SELECT count(*) FROM usuarios WHERE usuario='Cualquiera' AND password=' ' OR '1'='1';
```

El intérprete de SQL analiza la sentencia anterior en donde hay dos condiciones separadas por un OR. La primera condición no se cumplirá, pero la segunda se cumplirá siempre (1=1). Esta consulta devolverá el número de usuarios que hay en la tabla ya que la condición se cumple en todas las filas. Al ser el número mayor de 0, el atacante accedería al sistema.

Técnicas de explotación de SQL Injection

El punto anterior es un ejemplo básico de SQL *Injection* y normalmente se utiliza la cadena **' or '1'='1** como primera prueba para saber si la consulta es vulnerable ya que es la más sencilla. El problema radica en que esta cadena no funcionará en todas las consultas vulnerables o no nos permitirá obtener mucha más información de la base de datos por lo que si queremos explotar la vulnerabilidad debemos complicar el código inyectado. Para ello, existen diferentes tipos de técnicas con las que se puede explotar una vulnerabilidad SQL *Injection.* Se dividen en dos grupos: ataques SQL *Injection* y *Blind* SQL *Injection*.

SQL *Injection*, es la forma más básica de ataque y se da cuando el atacante es capaz de obtener los datos por pantalla. Esta condición no será posible en la gran mayoría de los casos, pero es la forma más rápida de obtener la información de una base de datos. Si no es posible obtener la información por pantalla cabe la posibilidad de obtener todos los datos a través de ataques ***Blind* SQL *Injection***. Estos ataques 'a ciegas' se basan en obtener información de la base de datos a través del comportamiento de la aplicación ante inyecciones de código. Para ello

debemos detectar un parámetro vulnerable y observar la respuesta de la aplicación a ciertas inyecciones.

Dentro de las inyecciones vamos a diferenciar dos tipos de comportamiento: inyecciones de comportamiento cero (ISQL0) e inyecciones de comportamiento positivo (ISQL+). Con ISQL0 la aplicación deberá comportarse como si no se hubiera inyectado código, mientras que con una ISQL+ deberá cambiar (mostrar un error, cambiar de página, etc.).

Fijándonos en la salida del sistema con estos tipos de inyecciones podemos desarrollar una lógica que nos permita obtener casi cualquier dato de la base.

Dentro de cada grupo de inyecciones SQL existen diferentes técnicas [6] para explotar la vulnerabilidad. Las que vamos a analizar por compatibilidad con sqlmap [4] son: ***boolean-based blind***, ***time-based blind***, ***error-based***, ***UNION query-based***, ***stacked queries*** y ***out-of-band***.

Para explicar las diferentes técnicas vamos a escoger una consulta a una base de datos.

```
SELECT alias FROM usuarios WHERE Id='$datosRecibidos';
```

Partiendo de esta base, vamos a mostrar cómo quedaría la consulta después de cada técnica para ilustrar su metodología.

2.1.1. Boolean-based blind

Esta técnica se puede utilizar cuando la aplicación sólo es vulnerable *Blind SQL Injection*. Se basa en obtener los comportamientos positivos y negativos de la aplicación. Esto significa que se inyectará un código al parámetro vulnerable que provocará que la aplicación no cambie (ISQL0) o que la aplicación cambie (ISQL+).

```
Ejemplo de ISQL0: ’ and ‘1’=’1
Ejemplo de ISQL+: ’ and ‘1’=’2
```

Estos dos ejemplos son básicos y se puede ver claramente como cambiaría el resultado si la inyección de cada uno tuviera éxito. Para obtener información con este tipo de inyecciones se pueden inyectar las siguientes funciones: Substring(texto, inicio, longitud), ASCII(cadena) y Lenght(texto).

Con estas tres funciones podemos componer una consulta que devolverá verdadero o falso si existe en una determinada posición de la base de datos una cadena de una determinada longitud con un valor ASCII dado. Si la aplicación se comporta como ISQL0 significará que sí que existe. Por ejemplo, si conseguimos que la consulta resulte de la siguiente forma:

```
SELECT alias FROM usuarios WHERE Id='1' AND ASCII(SUBSTRING(alias,1,1))=97 AND '1'='1'
```

Si se comporta como ISQL0 significará que existe un alias que en la posición (1,1) tiene un carácter con el valor ASCII=97. Esta comprobación se realiza para todos los valores ASCII y para todos los caracteres del alias, lo que provoca una necesidad de realizar una cantidad ingente de inyecciones.

2.1.2. Time-based blind

Los ataques ***Time-based Blind*** son similares a los *boolean-based blind*. Se utilizan cuando la aplicación es vulnerable a *Blind SQL Injection*.

Se trata de estudiar la forma en que se comporta la aplicación ante inyecciones, pero en este caso, en vez de fijarnos en la respuesta de la aplicación, nos vamos a fijar en el tiempo de las respuestas. Se inyectarán comandos que provocarán retrasos en las respuestas de la base de datos, si la respuesta tarda más que el retraso inyectado significará que el comando se ejecuta y por tanto las condiciones que introdujimos son correctas. Se suele utilizar la función 'IF' para implementar los retardos en las respuestas. Por ejemplo, consiguiendo la siguiente consulta:

```
SELECT alias  FROM usuarios WHERE Id='10' AND IF(version() like '5%', sleep(10), 'false'))--
```

Si la consulta tarda más de 10 segundos significará que la versión de la base de datos es 5.X (en MySQL). Al igual que en la técnica *boolean-based blind*, es necesario un gran número de inyecciones para obtener los datos.

2.1.3. Error-based

Este tipo de técnica es diferente a las anteriores. Se utiliza cuando el atacante no puede sacar información a través de técnicas como *UNION query-based* en entornos de SQL *Injection*. La idea principal de esta técnica es provocar un error en la base de datos, que lo muestre por pantalla y obtener algún dato dentro de ese error.

2.1.4. UNION query-based

Este tipo de técnica se puede realizar cuando podemos obtener los datos por pantalla. La idea general es insertar el comando UNION para añadir una nueva consulta (diseñada por el atacante). La aplicación mostrará por pantalla las dos consultas y obtendremos la información que queramos. Por ejemplo:

```
SELECT alias, password, dni FROM usuarios WHERE Id='10' UNION ALL SELECT direccion,1,1 FROM usuarios
```

El mayor problema para efectuar esta técnica es saber el número de columnas de la primera consulta para evitar errores de sintaxis al introducir la segunda. En este ejemplo es evidente ya que sabemos cómo es la consulta resultante, pero en un ataque real no dispondremos de esta información sin realizar unos pasos previos. Una de las técnicas para obtener el número de columnas de la consulta es mediante la inyección del comando 'ORDER BY'. Por ejemplo:

```
SELECT alias, password, dni FROM usuarios WHERE Id='10' ORDER BY 3 --
```

Si la consulta tiene éxito querrá decir que la consulta tiene 3 o más columnas. Solamente es hacer pruebas hasta dar con el valor exacto.

2.1.5. Stacked queries

Esta técnica es muy parecida a la anteriormente vista. La gran diferencia es que no se limita solamente a obtener datos mediante una consulta SELECT sino que se puede ejecutar cualquier tipo de comando, con la gravedad que eso supone. Así por ejemplo:

```
SELECT alias, password, dni FROM usuarios WHERE Id='10'; Drop table usuarios;
```

Este tipo de técnica introduce un final de comando ';' permitiendo al atacante introducir el comando que él quiera. La inyección anterior borraría la tabla 'usuarios', por lo que podríamos catalogar esta técnica como la más peligrosa.

2.1.6. Out-of-band

Esta técnica es similar a *error-based*, pero en un entorno solo compatible con *Blind SQL Injection*. La técnica se basa en utilizar comandos del gestor de bases de datos para enviar los resultados a un lugar externo (escribir en un texto, enviar por HTTP, etc.). En MySQL se puede inyectar el comando 'INTO OUTFILE', por ejemplo:

```
SELECT alias, password, dni FROM usuarios WHERE Id='10' INTO OUTFILE '/rutaWeb /query.txt';
```

Para realizar esta técnica se deben tener los permisos necesarios.

¿Cómo protegerse?

Aunque el abanico de técnicas es amplio, todas tienen la misma base: interpretar los datos recibidos como código. Para evitar estos ataques por tanto, es crucial evitar esta interpretación, y para ello existen tres técnicas que nos ayudarán a evitarlo [7].

- **Consultas parametrizadas:** Esta es la forma más sencilla de evitar ataques *SQL Injection*. Es fácil y rápida de implementar, y más fácil de interpretar. Utilizando esta técnica nos aseguramos que el intérprete diferencie entre código SQL y los datos recibidos. En este proyecto he utilizado este tipo de consultas para evitar SQL *Injection* en la aplicación principal. Por ejemplo, he utilizado esta sentencia para obtener el alias de un servidor:

```
$stmt = $dbConnection->prepare("SELECT serverAlias FROM servers WHERE id_server = ?");

$stmt->bind_param("d", $serverId);

$stmt->execute();
```

La clase utilizada con PHP para la gestión de las consultas es *mysqli*.

- **Procedimientos almacenados:** Es un sistema muy similar al de consultas parametrizadas, en donde, primero se crea la consulta y luego se añaden los datos en el momento de llamar a la base de datos. La diferencia radica en que en este caso las consultas están almacenadas previamente en la base de datos y son llamadas desde la aplicación.
- **Escapar todos los datos recibidos:** Al aplicar un escapado a todos los datos también conseguimos que no se interpreten como código. Aun así, es la opción menos recomendable de las tres, ya que existen técnicas de ofuscación de los datos que permiten saltarse el escapado y finalmente conseguir inyectar el código. Cada base de datos tiene sus propias reglas de escapado, por lo que hay que adaptarse a cada caso.

Otras prácticas recomendables son:

- **Otorgar privilegios mínimos**. Puede evitar ataques como *out-of band o stacked queries.*
- **Utilizar validación positiva de los datos.** Comprobar longitud, tipo de dato e identificar los tipos de entradas aceptadas. Se pueden añadir las comprobaciones que se consideren necesarias.

2.2. Sqlmap

Como podemos observar, los ataques SQL pueden ser varios y bastante complejos dependiendo del nivel de seguridad de la base de datos objetivo. Es por ello que se necesitan probar varias consultas y basándose en los resultados considerar los puntos débiles de la aplicación. Esto deriva en que implementar un sistema capaz de detectar las vulnerabilidades de inyección SQL en un servidor externo puede llevar mucho tiempo que repercutiría en el desarrollo y objetivos del TFM. Como se nos ha permitido utilizar herramientas de terceros que hagan ese trabajo he decidido utilizar **sqlmap**.

Sqlmap es un programa de código abierto escrito en Phyton que realiza un escáner de vulnerabilidades de inyección SQL con unos parámetros indicados por el usuario. El programa se encarga de probar diferentes inyecciones para detectar si el parámetro es vulnerable, de más o menos complejidad. Es capaz de detectar si un parámetro es vulnerable a cualquier tipo de técnica vista en el apartado anterior.

La forma de funcionar de sqlmap es a través de línea de comandos en donde el usuario debe introducir los parámetros que quiere escanear. Este punto es importante a la hora de realizar este proyecto. Sqlmap no es capaz de entrar en un servidor y devolver todas las vulnerabilidades SQL sin la participación activa del usuario. Dos parámetros básicos son la URL del servidor y la cookie de sesión en caso de que los parámetros se encuentren en una zona protegida.

Por otro lado, Sqlmap diferencia entre parámetros GET y POST, por lo que es necesario ejecutar diferentes comandos dependiendo del tipo de solicitud. Es recomendable utilizar algún tipo de proxy antes de realizar un escáner para obtener el nombre exacto de los parámetros dentro de la aplicación. Mi recomendación personal es utilizar Tamper Data, para navegadores Firefox.

Cabe la posibilidad de que sqlmap reconozca los parámetros necesarios, a través del comando '-- form'. El problema es que al utilizar dicho comando no se pueden dar valores a los parámetros enviados. Esto es un problema para detectar vulnerabilidades *Blind* SQL. Como dijimos anteriormente, en dichos ataques es necesario comparar dos comportamientos de la aplicación, uno nulo y otro positivo. Si enviamos un parámetro con el valor correcto, sqlmap será capaz de obtener los dos comportamientos más fácilmente y así detectar un mayor número de vulnerabilidades tipo *Blind*.

Imagen 2 – Sqlmap

Es por ello que he decidido que en este TFM se implementará un sistema que permita al usuario realizar los análisis de tres formas diferentes. A través de GET o POST con valores escogidos por el usuario para cada parámetro o a través del comando 'form' sin necesidad de introducir el nombre de los parámetros o su valor.

3. Estructura lógica de la aplicación

La aplicación se dividirá en cuatro páginas: Login, Main, Recod e Info. Cada página tiene unos procesos[1] específicos que realizará para cumplir sus objetivos. Por otro lado, la aplicación utiliza una sola base de datos en donde se almacena toda la información.

Página Login

Los objetivos de la página Login son: dar o denegar el acceso a los usuarios mediante un sistema usuario/contraseña y dar de alta a nuevos usuarios. Para ello utilizo dos procesos diferentes:

- Proceso 'login': Consulta en la base de datos si existe el par alias/contraseña enviados por el usuario. Si existen dentro del sistema damos acceso al usuario a la página Main.
- Proceso 'logup': Dar de alta al usuario con el alias y la contraseña enviados. Comprueba que el alias no existe en la base de datos y si los parámetros son correctos se guardan. Al guardar los datos se cifra la contraseña del usuario.

Para acceder a todas las páginas posteriores a la página Login será necesario una autenticación previa. Si no se encuentra una sesión abierta, la página redirigirá al usuario a la página Login.

Página Main

Los objetivos de la página Main son: crear un formulario que permita introducir y enviar los datos necesarios para efectuar un análisis. Una vez enviados los datos se ejecutarán los siguientes procesos.

- Proceso 'Escáner': Su objetivo es el de obtener los datos enviados por el usuario, realizar un estudio de si son correctos y ejecutar el comando en sqlmap correspondiente. En caso de no ser correctos se enviará un mensaje a la página Main informando del error ocurrido. En este proceso también se almacenan los datos del servidor en caso de que el servidor no estuviera dado de alta anteriormente.
  Los datos enviados para efectuar un análisis son: URL del servidor, alias del servidor, ruta del parámetro, cookie y opción de escaneo automático o los parámetros con sus respectivos valores según el protocolo elegido.
- Gestión de resultados: Los resultados obtenidos tras la ejecución de sqlmap no están en un formato cómodo para almacenarlos con vistas a un tratamiento posterior. Es por ello que es necesario tratar los datos obtenidos.
- Proceso 'saveScanner': Este proceso se conecta con la base de datos para almacenar los resultados del escáner realizado.

Si se efectúan todos los procesos correctamente se redirigirá al usuario hacia la página Record, que deberá de enseñar al usuario su último análisis realizado.

Página Record

Los objetivos de esta página serán los de mostrar al usuario sus escáneres realizados. La página recibirá en su URL dos parámetros: servidor y escáner, que indicarán el servidor que quiere consultar y el número de escáner dentro de ese servidor.

[1] Los procesos no se tienen porque relacionar directamente con una función o clase, sino que pueden ser un conjunto de ellas que realicen los objetivos marcados.

- Proceso ‘getScanners’: Este proceso obtendrá los valores de servidor y escáner y realizará una consulta a la base de datos en donde obtendrá los resultados correspondientes a los parámetros recibidos. Una vez obtenidos los representamos en la página Record.

Los resultados se mostrarán en formato de tabla para facilitar su lectura al usuario.

Página Info

La página Info es la más sencilla de todas. Solamente se enseña un texto con una descripción del proyecto y cómo funciona.

Para una visión más clara y global de todas las páginas y procesos se puede consultar la Ilustración 2.

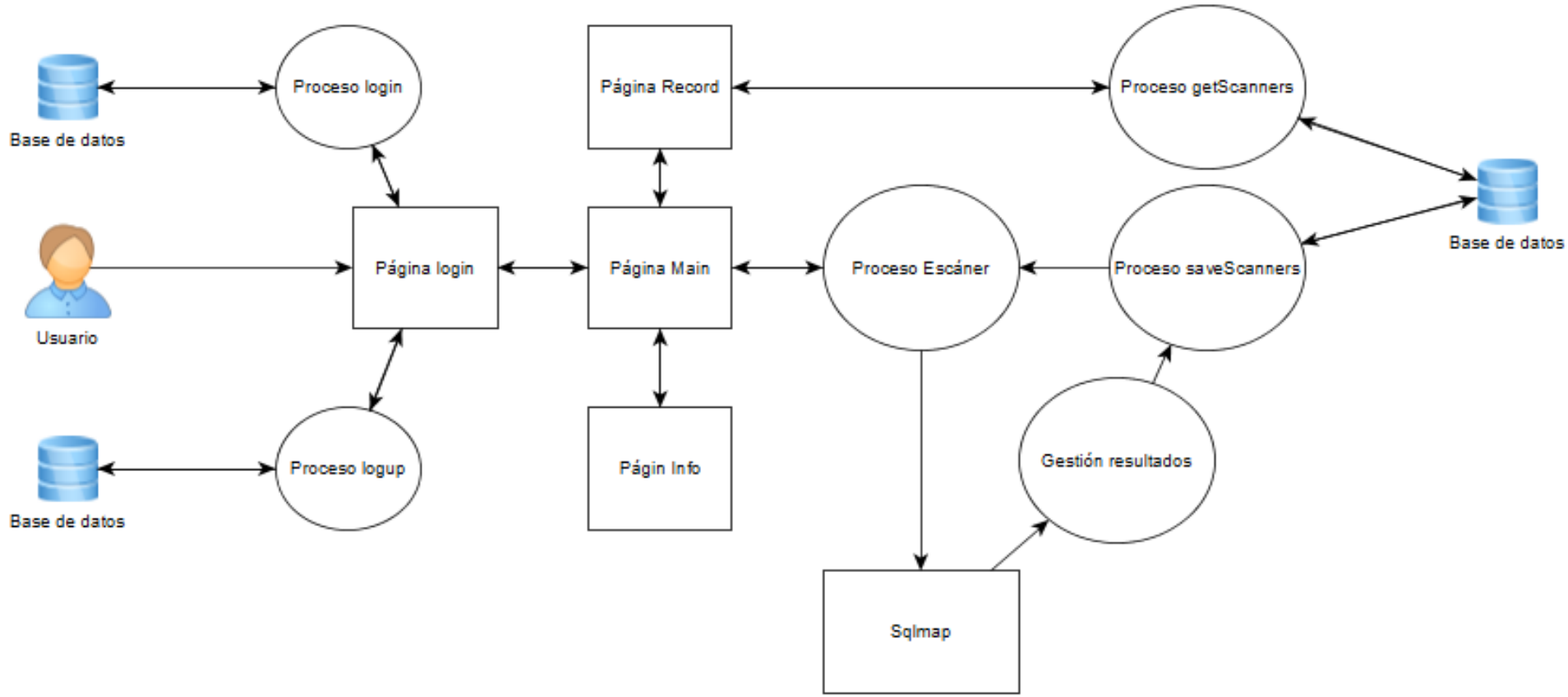

Ilustración 2 - Diagrama lógico de la aplicación

4. Symfony y estructura del código

Aunque no entre dentro de los objetivos del TFM, me ha parecido importante hacer una pequeña introducción a Symfony y así poder justificar la estructura de la aplicación desarrollada.

El *framework* Symfony divide las aplicaciones en *bundles*. Un *bundle* se puede considerar como un *plugin* o una división suficientemente grande de la aplicación como para desarrollarla en un módulo diferente. Dentro de cada *bundle* existen controladores. Los controladores son archivos PHP que se ejecutan si se dan ciertas condiciones (acceso a una URL, peticiones HTTP, etc.). En la aplicación final solamente he utilizado un *bundle*, dentro del cual se encuentran una serie de controladores que llevan a cabo el trabajo del *back-end*.

La estructura de una aplicación en Symfony es la siguiente:

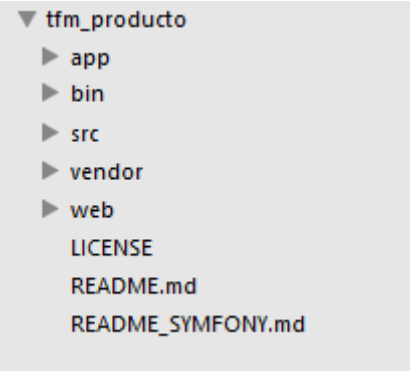

Imagen 3 - Estructura base de la aplicación

La carpeta “app” es la encargada de la configuración general de la aplicación así como gestionar y almacenar la cache y los archivos log. Dicha carpeta también se puede utilizar para almacenar archivos que se utilicen en varios controladores o *bundles.* Las carpetas “bin” y “vendor” se encargan de gestionar las dependencias externas instaladas. La carpeta “web” es donde se almacena la información pública como imágenes, archivos CSS o JavaScript. Por otro lado en esta carpeta se encuentra el archivo ‘robots.txt’ así como los archivos PHP del inicio de la aplicación (‘app_dev.php’ y ‘app.php’), dependiendo de si se encuentra en el entorno de desarrollo o de producción. Por último tenemos la carpeta “src”, en la que se almacena todo el trabajo de la aplicación (*bundles*, controladores, rutas, clases PHP, etc.). En este proyecto la mayoría del trabajo se ha realizado en esta carpeta por lo que es de vital importancia. La estructura final de esta carpeta es la siguiente:

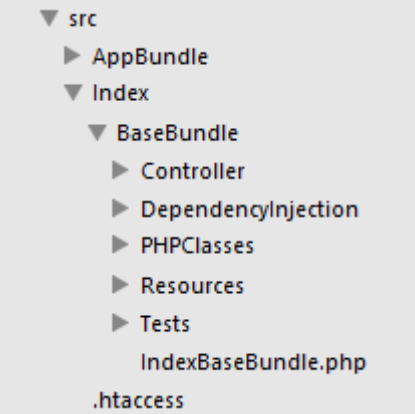

Imagen 4 - Carpeta src

La primera división de la carpeta hace referencia a los *bundles.* En este caso tenemos AppBundle que se instala por defecto con Symfony para facilitar la gestión de la aplicación en el entorno de desarrollo e IndexBaseBundle que es el *bundle* utilizado en la aplicación del TFM. Centrándonos en el *bundle* de la aplicación vemos que se divide en varias carpetas. Las más importantes son “Controller” y “Resources”. La carpeta “Controller” es la encargada de almacenar los controladores de la aplicación, por otro lado, la carpeta “Resources” almacena los archivos correspondientes al *front-end.*

Symfony ofrece dependencias de terceros para facilitar ciertas tareas como pueden ser Doctrine para las bases de datos y su propio sistema de gestión de sesiones. Ambos sistemas necesitan un plazo de aprendizaje el cual penalizaría el desarrollo del TFM por lo que he optado por no utilizarlos. La única dependencia de terceros utilizada ha sido Twig [11], para facilitar la utilización de código PHP en los documentos HTML.

Con esta pequeña introducción a Symfony ya podemos pasar a explicar el producto desarrollado y todas las funcionalidades implementadas para alcanzar los objetivos del TFM.

5. Diseño e implementación *front-end*

El *front-end* en esta aplicación se ha desarrollado en base a los objetivos marcados del TFM. Se dividirá en cuatro páginas principales: Login (acceso), Main (escáner), Info (información de la aplicación) y Record (historial de escáneres). Cada parte está compuesta por uno o varios archivos 'html.twig', que marcan el contenido; un archivo CSS, que indica el estilo y la distribución; y un apartado JavaScript para las comunicaciones y modificaciones dinámicas de la página.

He decidido que las comunicaciones con el servidor se realicen con AJAX (*framework* jQuery). Una de las ventajas de realizar este tipo de comunicaciones desde JavaScript es que facilita la gestión de los resultados. Los datos de envío están en formato HTML clásico, pero las respuestas son en formato JSON para que sean gestionadas de forma más sencilla en el *front-end*.

A continuación explico con detalle cómo es el *front-end* de cada página de la aplicación.

5.1. Página Login

La página Login es la primera página que ve el usuario al entrar a la aplicación. Gracias al sistema de rutas implementado en el *back-end* (explicado en el próximo capítulo) si un usuario no está autentificado se redirige automáticamente hacia esta página. Cuenta solamente con un fichero 'html.twig', en donde se hace toda la distribución, su nombre es 'login.html.twig'. El aspecto visual de la página es el siguiente.

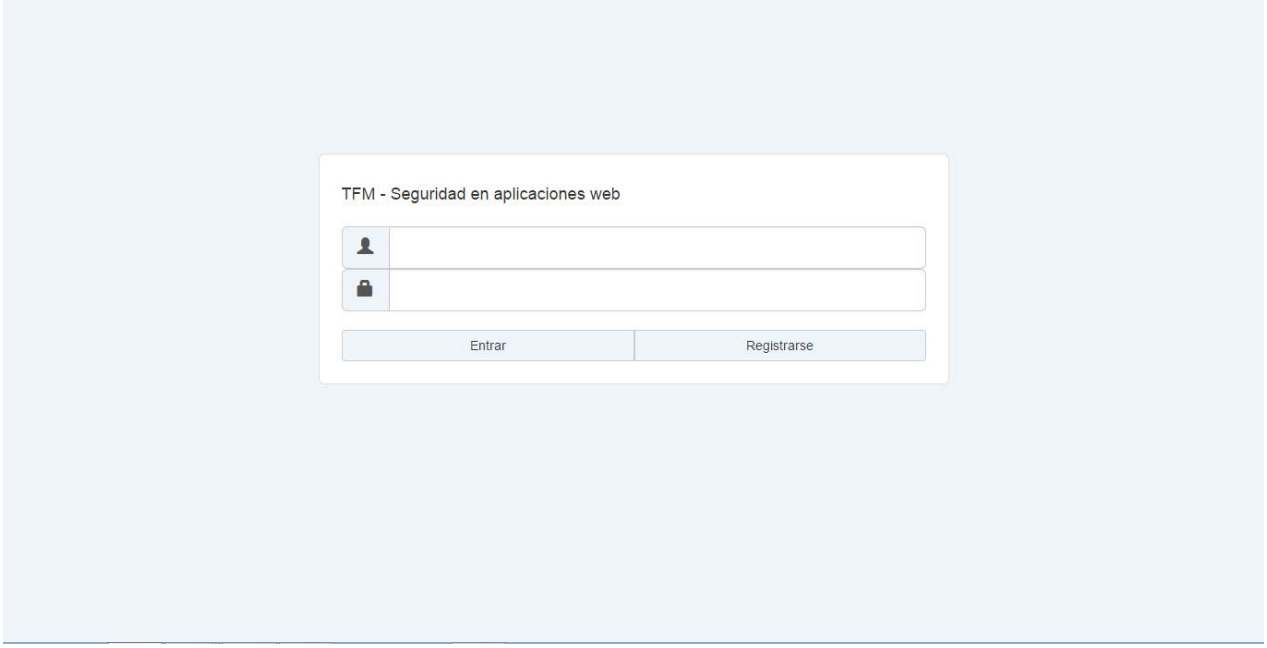

Imagen 5 - Página login

Como podemos observar, en la parte central tenemos un formulario de entrada a la aplicación. En él nos permite introducir nuestras credenciales o darnos de alta en la aplicación a través del mismo formulario. La distribución de la página se ha realizado mediante CSS con la función "flex". Se han realizado tres divisiones verticales con una relación de 2 a 1 a favor de la división central.

Dentro de la división central se han introducido otras tres divisiones horizontales, con la misma relación. En el punto central se ha introducido dos *input-groups*[2], uno para el alias y otro para la contraseña, y abajo se han introducido dos *btn-group*, uno para acceder y otro para registrarse. Debajo de los *btn-groups* se encuentra una división solamente visible si se produce algún error.

La información enviada al servidor son dos peticiones POST: 'login' y 'logup'. En caso de que no ocurra un error en la petición 'login' se redirigirá automáticamente hacia la página Main. Si no ocurre ningún error dentro de la petición 'logup', aparecerá un mensaje en pantalla informando que el usuario ha sido dado de alta correctamente.

5.2. Barra de navegación

Una vez dentro de la aplicación principal he decidido crear un apartado fijo que será común a las tres páginas restantes. Este apartado hará la función de barra navegadora y nos permitirá movernos fácilmente a través de las diferentes páginas de la aplicación. Está compuesta por dos páginas HTML diferentes: 'base.html.twig' y 'nav.html.twig'. La página 'base' genera la estructura de la página mientras que la página 'nav' añade la barra de navegación.

Imagen 6 - Barra de navegación

Al declarar cualquier página como 'hija' de 'nav', su contenido se imprimirá en el cuadrado grande, quedando la barra de navegación inalterada.

Cada vez que se accede a una de las tres páginas de contenido, se comprueba la dirección solicitada y según el resultado resaltará el botón correspondiente a la página en la que se encuentra el usuario. En este caso no se envía ningún dato al servidor ya que su función es meramente visual.

[2] Tanto los *input-groups* como los *btn-groups* son clases CSS definidas dentro de Bootstrap.

5.3. Página Main

La página Main es la encargada de recoger la información necesaria para realizar un análisis en un servidor externo. Está compuesta por un formulario para enviar los datos.

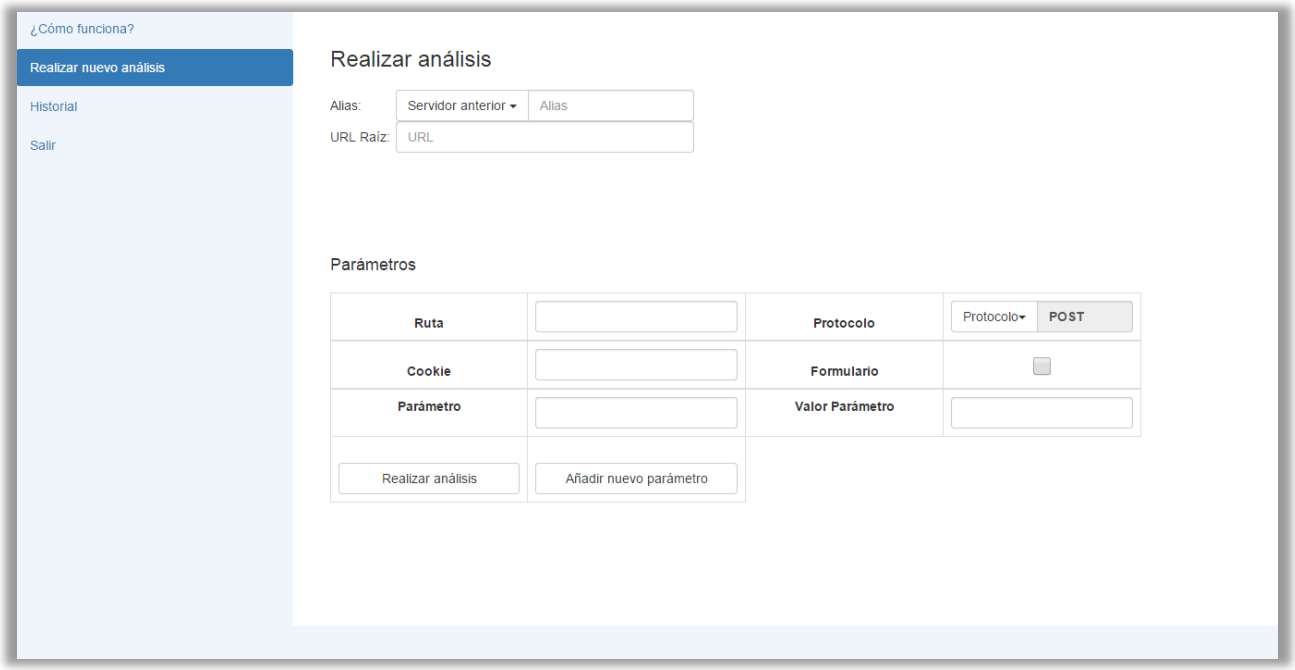

Imagen 7 - Página Main

El contenido de la página se divide en dos divisiones 'flex' orientadas verticalmente con proporción 2 a 1 en favor de la división inferior.

En la parte superior realizamos una división horizontalmente con relación 2 a 1 en favor de la división que se encuentra más a la derecha de la pantalla. En la división izquierda encontramos un pequeño formulario en donde el usuario debe insertar el alias del servidor (a elección del usuario) y la URL del mismo. Dicho formulario es dinámico y cambia según los servidores consultados por el usuario anteriormente. En el *input* del alias encontramos un *dropdown-menu.* Este *dropdown-menu* desplegará la lista de los servidores consultados por el usuario anteriormente. Si aún no ha consultado ninguno se desactiva el botón y el usuario puede introducir un servidor nuevo. Para obtener los servidores procedemos de la siguiente manera:

- Cuando se carga la página se recibe un *array* mediante PHP con los Ids de los servidores consultados anteriormente. Se comprueba la longitud del *array* y si es igual a 0 se desactiva el botón del *dropdown-menu*.
- Si la longitud del *array* recibido es mayor que 0, se solicita al servidor la lista de alias que corresponden con los Ids recibidos mediante una petición tipo POST que he denominado 'getAlias'. Cuando recibimos los alias creamos el *dropdown-menú* con ellos.
- El *dropdown-menu* consistirá en una lista de botones con el alias de los servidores consultados por el usuario anteriormente. Cuando el usuario pulse el botón de cualquiera de ellos se solicitará al servidor su alias y su URL mediante una petición POST que he denominado 'getURL'. Una vez recibidos, la página actualizará los *inputs* de alias y URL para que el usuario no tenga que introducirlos manualmente.

En la parte derecha de esta división se encuentra un espacio vacío. Este espacio se llenará con un mensaje cuando se ejecute un análisis para informar al usuario que hay un análisis en curso.

La parte inferior de esta página corresponde a la segunda parte del formulario que el usuario debe rellenar para realizar un análisis, estando directamente relacionada con sqlmap. Como dijimos en el capítulo 2, sqlmap no realiza un análisis automático de todos los parámetros de la aplicación sino que es necesario que el usuario indique el tipo de parámetro y su valor. La única forma de que sqlmap realice un escáner automático es mediante el comando 'form', limitándonos a no detectar vulnerabilidades del tipo *Blind* en algunos casos. Es por ello que el formulario se adaptará dinámicamente según la petición del usuario.

En este TFM se consideran 3 opciones: peticiones POST, peticiones GET o escaneo automático.

- **Escaneo automático:** Se realiza pulsando el botón de "Escaneo automático". Dinámicamente desaparecerán los *inputs* de parámetro y valor de la parte inferior mediante JavaScript. Para desactivar dicha opción se puede volver a pulsar el botón de "Escaneo automático".
- **Peticiones GET o POST:** En la pestaña de "Protocolo" se selecciona la forma elegida. Una vez hecho esto, ambos análisis necesitan los mismos datos de envío. Se puede añadir el nombre del parámetro y un valor. Pulsando el botón "Añadir nuevo parámetro" se insertarán dos nuevos *inputs* dinámicamente, que serán una copia de los *inputs* de "Parámetro" y "Valor". He considerado el límite de poder enviar solamente dos parámetros para este tipo de escáner. Poner un límite infinito de parámetros no era viable, y a partir de dos parámetros a escanear el tiempo de escáner se incrementaba de forma considerable.

Por último, en este formulario es necesario introducir un campo "Ruta". Este campo hace referencia a la ruta en la que se encuentran los parámetros dentro del servidor. Por ejemplo, introducimos la URL http://www.tfmdlrtest.es/. Dentro de esa URL existirán diferentes rutas, por ejemplo 'login.php' o 'index.php'. El parámetro estará dentro de alguna de esas rutas por lo que el usuario tendrá que introducir la correspondiente.

Si pulsamos el botón "Realizar análisis" enviaremos todos los datos que introducimos mediante una petición POST, que he denominado 'scanner'. Cuando se pulsa el botón, en el espacio superior derecho aparecerá dinámicamente el mensaje "Realizando análisis…" mediante JavaScript. En caso de que ocurra un error o el parámetro no sea ejecutable, lo mostramos por pantalla. En caso de que el escáner se complete con éxito, redirigimos al usuario hacia la página Record para consultar los resultados.

5.4. Página Record

La página Record se divide en tres partes orientadas verticalmente. La parte superior es una pequeña cabecera que nos permitirá navegar entre los diferentes servidores que hemos consultado. La división central tendrá dos divisiones horizontales. En la de la izquierda aparecerá información de la URL, fecha del análisis y ruta. En la parte derecha aparecerán los botones para navegar entre los diferentes escáneres realizados a ese servidor. Por último, la parte inferior mostrará los resultados del escáner en formato tabla.

La página Record recoge dos parámetros mediante la URL, por ejemplo '/record/0/0'. Estos dos parámetros serán los utilizados para navegar entre los servidores y los escáneres de cada servidor. Cuando se abre la página se recibe del *back-end* un *array* con los alias de los servidores ordenados por orden del primer escaneo, por lo que la posición cero del *array* será el primer servidor escaneado. Sabiendo esto se hace una comprobación de la URL. El primer parámetro recibido por la URL es el servidor, mientras que el segundo es el escáner. Si abrimos la URL '/record/0/2' estaremos indicando a la aplicación que nos enseñe el tercer escáner del primer servidor (empezamos a contar en 0). Por tanto, una vez recibido el *array* creamos la cabecera (división superior) con los alias de los servidores y resaltamos el servidor que coincida con el parámetro recibido mediante la URL.

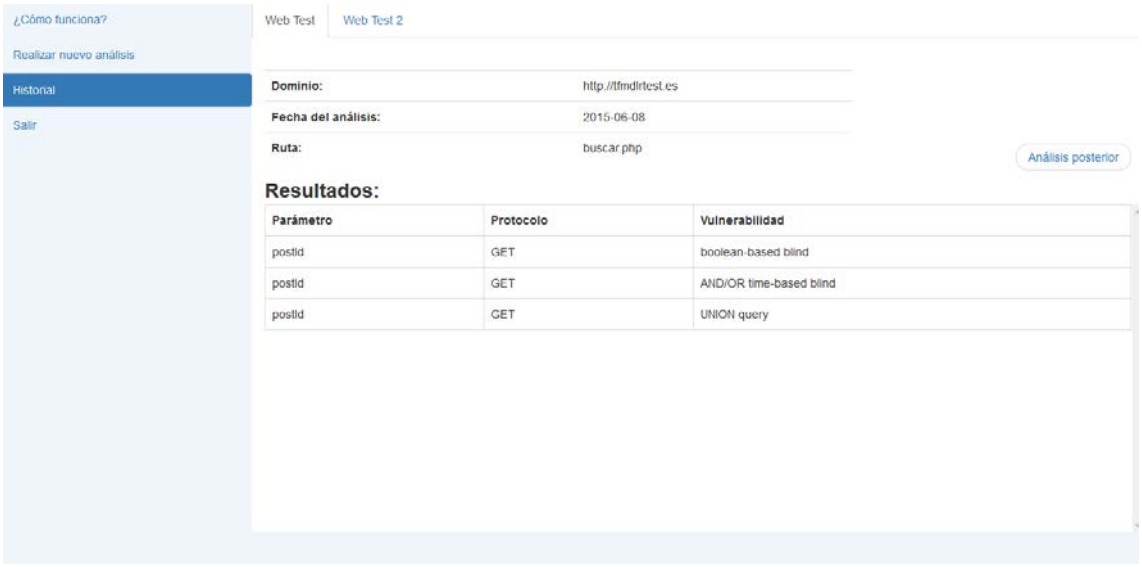

Imagen 8 - Historial

Una vez creada la cabecera se hace una petición para obtener los resultados del análisis seleccionado con la URL. Los análisis se seleccionan del mismo modo que los servidores, salvo que esta vez están ordenados al revés, por lo que el análisis con posición 0 de ese servidor corresponderá al último análisis realizado (y no al primero). La petición se realiza cada vez que se carga la página. Es una petición POST que he denominado 'getInfo'.

5.5. Página Info

Esta página es la más sencilla de la aplicación. Solamente es texto plano que nos indica una pequeña descripción de la aplicación y su funcionamiento.

Imagen 9 - Página Info

6. Implementación *back-end*

Las funciones del *back-end* son las de recibir y tratar peticiones que vienen del *front-end.* Antes de empezar con el listado de todas las consultas que se realizan al *back-end* en esta aplicación, vamos a enseñar la estructura de la base de datos con el fin de entender mejor el trabajo desarrollado en cada petición.

6.1. Base de datos

La DBMS escogida ha sido MySQL. Esta decisión se basa en mi preferencia personal ya que es con la que más cómodo me siento y con la que más he trabajado. El código SQL para la creación de la base de datos es la siguiente:

```
CREATE TABLE users(
id int AUTO_INCREMENT PRIMARY KEY,
alias varchar(25) NOT NULL,
password varchar(100),
UNIQUE (alias)
);

CREATE TABLE servers(
id_user int,
FOREIGN KEY (id_user) REFERENCES users(id),
serverAlias varchar(25) NOT NULL,
id_server int AUTO_INCREMENT PRIMARY KEY,
rootUrl varchar(50)
);

CREATE TABLE scanners(
id_server int,
FOREIGN KEY (id_server) REFERENCES servers(id_server),
id_scanner int AUTO_INCREMENT PRIMARY KEY,
route varchar(100),
currentDate date
);

CREATE TABLE results(
id_scanner int,
FOREIGN KEY (id_scanner) REFERENCES scanners(id_scanner),
parameter varchar(50),
protocol varchar(10),
vType varchar(100)
);
```

La base de datos se dividirá por tanto en cuatro tablas: ‘users’, ‘servers’, ‘scanners’ y ‘results’.

En la tabla ‘users’ se almacenarán el alias y el password de acceso de cada usuario. Cada uno tendrá un ‘id’ único que se utilizará como referencia en la tabla ‘servers’. La tabla ‘servers’ almacenará los datos básicos de un servidor como su URL, alias y usuario al que pertenece. Cada servidor tiene un ‘id’ único que se utilizará como referencia en la tabla ‘scanners’. La tabla ‘scanners’ almacenará la ruta y la fecha

de cada escáner realizado. La tabla ‘results’ almacenará los resultados de cada análisis. Los resultados se dividirán en parámetro, protocolo y tipo de vulnerabilidad.

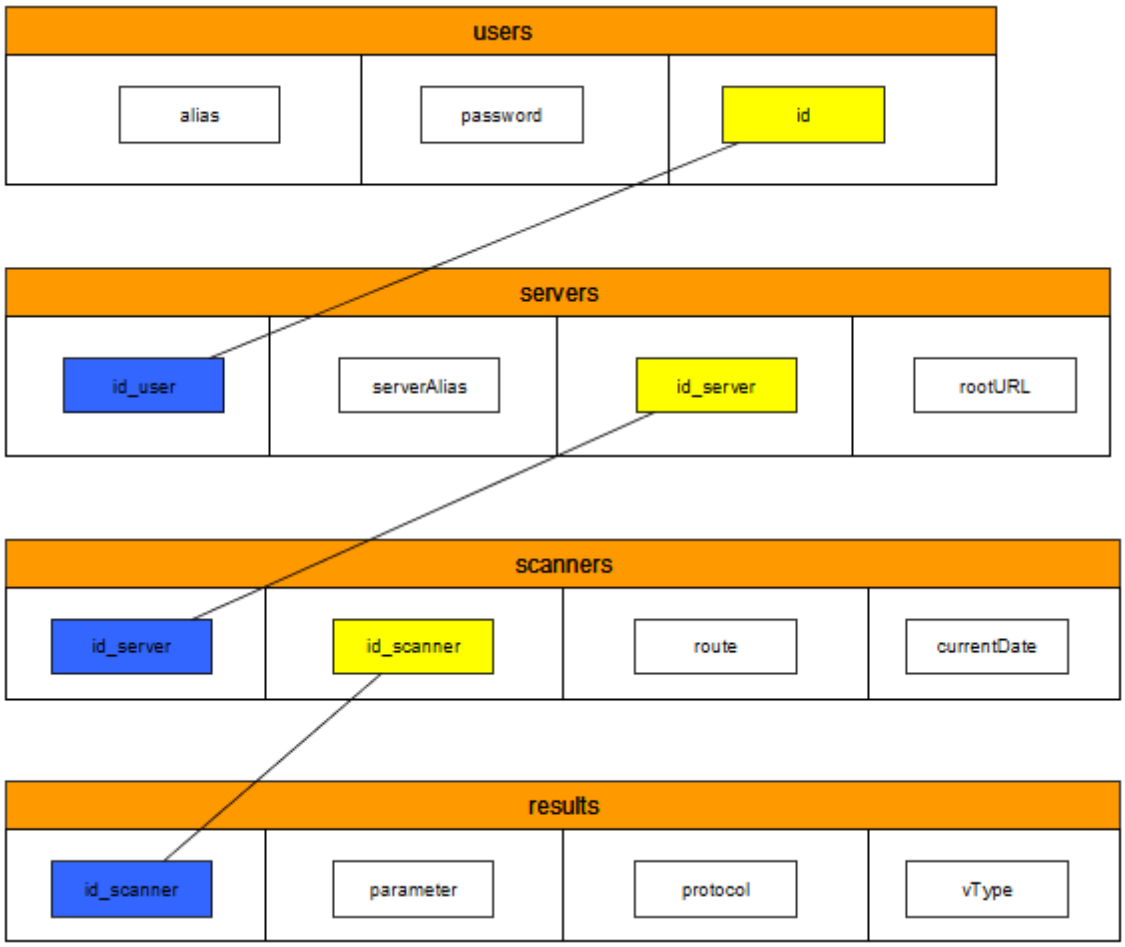

Ilustración 3 - Estructura de base de datos

En la ilustración anterior vemos la estructura de la base de datos de una forma más clara. Las líneas muestran la relación ‘Primary Key’ (cuadrados amarillos) y ‘Foreign Key’ (cuadrados azules) entre las diferentes tablas.

Gracias a esta implementación se ha logrado que cada usuario pueda tener varios servidores, que esos servidores puedan tener varios escáneres cada uno, y que cada escáner pueda tener varios resultados.

El usuario de la base de datos utilizado solamente tiene los privilegios de INSERT y SELECT:

```
GRANT SELECT ON tfm.* TO 'userTFM'@'localhost';
GRANT INSERT ON tfm.* TO 'userTFM'@'localhost';
```

Se le han asignado privilegios mínimos al usuario para evitar los posibles problemas de seguridad que pueda acarrear tener privilegios que no se llegan a utilizar.

6.2. Rutas y controladores

Como dijimos en el capítulo 2, Symfony se divide en *bundles* y *controllers*. Dentro de cada *bundle* se reparten distintos controladores encargados de realizar una o varias funciones. Para gestionar todos los *controllers* dentro de un mismo *bundle* se utiliza un archivo 'routing.yml' que se encuentra dentro de la carpeta '/Resources/config' del *bundle*.

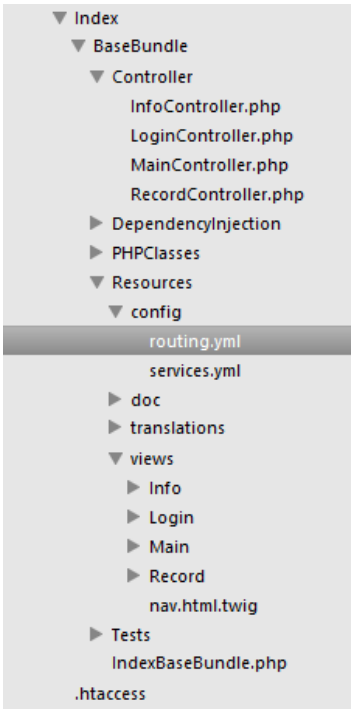

Imagen 10 – Archivo Routing.yml

Dentro de ese archivo están almacenadas todas las peticiones posibles al servidor. Cada petición es asignada a un controlador y una función que se ocupará de gestionarla. He decidido crear cuatro controladores, uno para cada página de la aplicación.

El primer conjunto de peticiones del archivo 'routing.yml' son las peticiones GET del contenido de las cuatro páginas:

```
index_base_homepage:
    pattern:     /
    defaults: { _controller: IndexBaseBundle:Main:index }
    methods:  [GET]

index_base_infopage:
    pattern:     /info
    defaults: { _controller: IndexBaseBundle:Info:index }
    methods:  [GET]

index_base_recordpage:
    pattern:     /record/{server}/{scanner}
    defaults: { _controller: IndexBaseBundle:Record:Index, server:0, scanner:0 }
    methods:  [GET]
```

```
index_base_loginpage:
    pattern:    /login
    defaults: { _controller: IndexBaseBundle:Login:index }
    methods:  [GET]
```

Esta parte del archivo nos asegura que la aplicación nos devolverá el archivo HTML correspondiente a cada página si navegamos hacia la ruta marcada en 'patterm'.
Si nos fijamos en el campo 'defaults' podemos ver el controlador y función asignada. Por ejemplo, si tenemos 'IndexBaseBundle:Main:index' significa que la petición será gestionada por el *bundle* 'IndexBase', con el controlador Main y la función 'index'. Si se trata de una consulta para obtener una página HTML, la función siempre se llamará 'index'.

Las siguientes líneas corresponden a las peticiones POST, en las que se produce un intercambio de información entre el *front-end* y el *back-end.*

```
index_base_getInfoAction:
    pattern:     /getInfo
    defaults: { _controller: IndexBaseBundle:Record:getInfo }
    methods:  [POST]

index_base_loginAction:
    pattern:    /login
    defaults: { _controller: IndexBaseBundle:Login:login }
    methods:  [POST]

index_base_logupAction:
    pattern:    /logup
    defaults: { _controller: IndexBaseBundle:Login:logup }
    methods:  [POST]

index_base_getURLAction:
    pattern:    /getUrl
    defaults: { _controller: IndexBaseBundle:Main:getURL }
    methods:  [POST]

index_base_getAliasAction:
    pattern:    /getAlias
    defaults: { _controller: IndexBaseBundle:Main:getAlias }
    methods:  [POST]

index_base_scannerAction:
    pattern:    /scanner
    defaults: { _controller: IndexBaseBundle:Main:scanner }
    methods:  [POST]
```

Estas líneas se interpretan de igual manera que las anteriores. Después de ver el archivo 'routing.yml' podemos hacer un resumen del trabajo realizado por el *back-end*:

Controlador Login

- Peticiones GET
  - /login. Gestionada por la función ‘index’.
- Peticiones POST
  - /login. Gestionada por la función ‘login’
  - /logup. Gestionada por la función ‘logup’

Controlador Main

- Peticiones GET
  - /. Gestionada por la función ‘index’.
- Peticiones POST
  - /getURL. Gestionada por la función ‘getURL’.
  - /getAlias. Gestionada por la función ‘getAlias’.
  - /scanner. Gestionada por la función ‘scanner’.

Controlador Record

- Peticiones GET
  - /record/{server}/{scanner}[3]. Gestionada por la función ‘index’.
- Peticiones POST
  - /getInfo. Gestionada por la función ‘getInfo’.

Controlador Info

- Peticiones GET
  - /info. Gestionada por la función ‘index’.

En los siguientes apartados vamos a explicar en detalle las funciones que realiza cada controlador siguiendo el resumen anterior. No vamos a entrar en detalle con el controlador Info, ya que solamente comprueba que el usuario está autentificado y en caso afirmativo envía la página al usuario, en caso negativo lo redirige a la página Login.

6.3. Controlador Login

El controlador Login posee tres funciones: ‘index’, ‘login’ y ‘logup’. Para facilitar el trabajo se ha desarrollado una clase PHP propia que se encargará principalmente de las consultas a la base de datos. La clase se encuentra en la carpeta PHPClasess dentro de la estructura del *bundle* y su nombre es DBClassLogin.

Función ‘index’

La función comprueba si el usuario que accede a la página ‘login’ está autenticado. De estarlo, destruye la sesión. Con esto conseguimos destruir la sesión cuando un usuario quiera salir de la aplicación desde otra página. Solamente lo redirigimos hacia la página Login y al cumplir la condición de estar autentificado, se destruye la sesión.

Acto seguido, envía la página ‘LoginBase.html.twig’ al usuario.

Función ‘login’

La función ‘login’ se ejecuta cuando el usuario realiza la petición POST ‘login’. Se encarga de verificar las credenciales de acceso del usuario, para posteriormente

[3] Recordemos que en la página Record se utilizan dos parámetros obtenidos mediante la URL.

concederle acceso o denegárselo. En dicha petición se envía el 'alias' y la 'password'. Una vez recogidos ambos parámetros se llama a la función 'login' de la clase desarrollada 'DBClassLogin'. En dicha clase se ejecuta el siguiente código:

```
$stmt = $dbConnection->prepare("SELECT alias,password FROM users WHERE alias = ?");

$stmt->bind_param("s", $user);

$stmt->execute();
$stmt->bind_result($aliasFetch, $passwordFetch);
while ($stmt->fetch()) {
    if(password_verify( $password , $passwordFetch )==true){
            return true;
    }else{
            return false;
            }
}
```

El código es una simple consulta a la base de datos preguntando si existe un usuario con el alias enviado. Si existe, comprobamos que el cifrado de la contraseña coincide con la contraseña cifrada almacenada en la base de datos mediante la función de PHP 'password_verify'. Si coincide devolvemos 'TRUE'.

Si la función anterior devuelve 'TRUE', creamos la sesión y enviamos un 'ok' al *front-end*. Cuando el *front-end* reciba dicho mensaje cambiará de página hacia la ruta '/' (página Main). En la sesión se guarda el alias del usuario para su utilización dentro de la aplicación.

Si la aplicación no devuelve 'TRUE', devolveremos '!ok' al *front-end* que mostrará un mensaje de error por pantalla.

Función 'logup'

La función 'logup' tiene como objetivo dar de alta a un usuario dentro del sistema. Para ello se reciben del usuario su 'alias' y su 'password'. Al igual que en la función anterior, utilizaremos la clase DBClassLogin para realizar las diferente tareas.

Una vez hecha la llamada a la función 'logup' de la clase mencionada, lo primero que hacemos es realizar la comprobación de si ese usuario ya está dado de alta en el sistema. Para ello utilizamos la función 'isThereUser', recibiendo como parámetro el alias del usuario en cuestión.

```
$stmt = $dbConnection->prepare("SELECT alias FROM users WHERE alias = ?");
$stmt->bind_param("s", $user);
```

Si la consulta no devuelve ningún resultado significará que no existe un usuario con ese alias, por lo que podemos proceder al alta.

```
$hashPassword = $this -> getHash($password);
$stmt = $dbConnection->prepare("INSERT INTO  users (alias,password) VALUES (?,?)");
$stmt->bind_param("ss", $user,$hashPassword);
```

La función ‘getHash’ es propia de PHP y nos permite cifrar la contraseña a almacenar.
Se le responderá al usuario con un mensaje dependiendo del resultado (alta correcta o usuario ya existente).

6.4. Controlador Main

El controlador Main posee cuatro funciones: ‘index’, ‘getAlias’, ‘getURL’ y ‘scanner’. Al igual que en el controlador Login, he desarrollado clases PHP para facilitar y organizar mejor el trabajo. En esta ocasión he utilizado dos clases diferentes: DBClassMain (consultas a la base de datos) y ClassMainFunctions (comprobación de parámetros, integrar sqlmap y tratamiento de datos). En este apartado no vamos a profundizar en la segunda clase ya que está relacionada directamente con sqlmap, del que hablaremos en el capítulo 7.

Función ‘index’
La función ‘index’ comprueba que el usuario está autentificado. De no estarlo lo redirige a la página Login. Si está autentificado se solicitan los Ids de todos los servidores utilizados por el usuario anteriormente mediante la función ‘getServersIds’ de la clase DBClassMain. Dicha función realiza la siguiente consulta:

```
$stmt = $dbConnection->prepare("SELECT servers.id_server FROM servers INNER JOIN users ON users.id = servers.id_user WHERE users.alias = ?");
$stmt->bind_param("s", $userAlias);
```

Donde ‘userAlias’ es el alias del usuario almacenado en la sesión. Una vez obtenido el *array* de Ids se envían al usuario junto a la página ‘MainBase.html.twig’.

Función ‘getAlias’
Esta función se ejecuta tras la petición del usuario ‘getAlias’. Recordemos que dicha función servía para obtener los alias de los servidores que el usuario había utilizado hasta el momento para rellenar el *dropdown-menú* de la parte superior de la página Main. El usuario recibe un *array* de servidores cuando se carga la página. Mediante JavaScript el *front-end* realiza un *loop* de peticiones ‘getAlias’ para obtener los alias de los servidores. Cuando recibe el alias crea el botón correspondiente del *dropdown-menu* dinámicamente.
Para obtener los alias se utiliza la función ‘getServerAlias’ de la clase DBClassMain. La consulta realizada es la siguiente:

```
$stmt = $dbConnection->prepare("SELECT serverAlias FROM servers WHERE id_server = ?");
$stmt->bind_param("d", $serverId);
```

La respuesta al usuario es el alias correspondiente.

Función ‘getURL’
Esta función se ejecuta cuando un usuario pulsa un botón del *drop-down menú*. Con ella se obtiene la URL del servidor seleccionado. Solamente se hace una consulta a la base de datos mediante la función ‘getServerUrl’ de la clase DBClassMain con el Id del servidor recibido.

```
$stmt = $dbConnection->prepare("SELECT rootUrl FROM servers WHERE id_server = ?");
$stmt->bind_param("d", $serverId);
```

Al usuario se le envía la URL del servidor y se muestra por pantalla el alias (almacenado de la función 'getAlias') y la URL del servidor seleccionado en el *dropdown-menu*.

Función 'scanner'

Esta función se ejecuta cuando un usuario solicita realizar un análisis. Recibe los siguientes parámetros: 'alias', 'url', 'route', 'cookie', 'sForm' y 'protocol'. Después se pueden enviar 'param' y 'paramValue', según la elección del usuario (al seleccionar 'Escaneo automático' no se envían). También cabe la posibilidad de enviar 'param1' y 'paramValue1', que corresponden al segundo parámetro enviado (si el usuario lo introduce).

El primer paso es verificar que todos los parámetros enviados son correctos (no están vacíos, no hay más parámetros de los deseados, etc.). Para ello utilizamos la función 'check' de la clase ClassMainFunctions. Si obtenemos algún error se lo enviamos al usuario. De lo contrario, el siguiente paso es comprobar si el alias y la URL del servidor enviado ya existen en la base de datos.

Para la comprobación he utilizado la función 'checkServerAlias'. En dicha función se hace la siguiente consulta:

```
$stmt = $dbConnection->prepare("SELECT count(*) FROM servers INNER JOIN users ON users.id = servers.id_user WHERE users.alias = ? AND servers.serverAlias=? AND servers.rootUrl= ? ");
$stmt->bind_param("sss", $userAlias, $serverAlias, $serverUrl);
```

Donde '$userAlias' es información guardada en la sesión. Los otros parámetros los envía el usuario. Si la consulta devuelve 0, significará que no existe ese servidor en la base de datos. En ese caso, almacenamos la información llamando a la función 'insertServer':

```
$stmt = $dbConnection->prepare("INSERT INTO servers (serverAlias, rootURL, id_user) SELECT ?,?, id FROM users WHERE users.alias=?");
$stmt->bind_param("sss", $serverAlias, $serverUrl, $userAlias);
```

Una vez terminado el proceso de comprobación es momento de interpretar los datos recibidos. Recordemos que en esta aplicación hay tres formas de enviar los datos: escaneo automático, parámetros con protocolo GET y parámetros con protocolo POST. En esta parte es momento de diferenciar cuál de ellos ha seleccionado el usuario. Para ello nos fijamos en el parámetro recibido 'sForm'.

Si el parámetro 'sForm' es verdadero significará que el usuario ha seleccionado 'Escaneo automático'. Si es falso, debemos observar el parámetro 'protocol' para saber si ha escogido GET o POST. Según los parámetros escogidos se deberá hacer un tipo diferente de consulta a sqlmap. Este punto es extenso y he decidido seguirlo en el capítulo 7 "Integración de sqlmap".

Imaginemos que ejecutamos los comandos de sqlmap correctamente y el escáner ha sido un éxito. El siguiente paso es enviar la confirmación al usuario. Para ello debemos enviar el índice que ocupa el servidor en la tabla de servidores del usuario para que el *front-end* pueda dirigir al usuario a la página correcta del historial. Para

ello, utilizamos la función ‘getServerKey’. La función completa realiza las siguientes acciones:

```
$stmt = $dbConnection->prepare("SELECT servers.serverAlias FROM servers INNER JOIN users ON users.id = servers.id_user WHERE users.alias = ?");

$stmt->bind_param("s", $alias);
$stmt->execute();
$array = array();
$stmt->bind_result($serverAlias);
while ($stmt->fetch()) {
    $array[] = $serverAlias;
    }
$stmt->close();
$dbConnection->close();
$key = array_search($serverAlias, $array);
return $key;
```

Obtenemos todos los alias de los servidores del usuario y devolvemos el índice donde se encuentra el servidor con ese alias. Devolvemos el índice (‘$key’) al *front-end* y redirigirá al usuario hacia la página ‘/record/$key/0’. La página Record sabrá qué servidor está seleccionando el usuario haciendo la misma operación, obteniendo el *array* de todos los servidores y escogiendo el de índice ‘$key’.

6.5. Controlador Record

El controlador Record posee dos funciones: ‘index’ y ‘getInfo’. Al igual que en los controladores anteriores, he desarrollado una clase PHP para facilitar y organizar mejor el trabajo. En esta ocasión he utilizado la clase DBClassRecord con la finalidad de gestionar las consultas a la base de datos.
Recordemos que en esta página recibimos dos parámetros: $server y $scanner mediante la URL.

Función ‘index’
Al igual que en la página Main, es necesario verificar que el usuario está autentificado para poder enviarle la página del historial. De no estarlo se redirige a la página Login. Si está autentificado es necesario realizar una serie de comprobaciones debido a los parámetros que se reciben en la URL. Recordemos que esos parámetros sirven para navegar por los diferentes servidores y escáneres del usuario por lo que un valor inadecuado puede generar errores.
La primera comprobación que realizamos tiene que ver con los servidores (primer parámetro recibido). Realizamos una llamada a la base de datos mediante la función ‘getServers’, que recibe el alias del usuario. Esta función nos devolverá un *array* con los servidores utilizados por ese usuario. Su código:

```
$stmt = $dbConnection->prepare("SELECT servers.serverAlias FROM servers INNER JOIN users ON users.id = servers.id_user WHERE users.alias = ?");
$stmt->bind_param("s", $alias);
```

Dependiendo de la longitud del *array* podemos llegar a las siguientes conclusiones:

- Si la longitud del *array* es 0 significará que el usuario no ha hecho ningún análisis todavía. En ese caso he decidido reenviar al usuario a la página Main.

- Si $server es menor que cero causará un error al intentar acceder a la posición <0 del *array* de servidores, por lo que he decidido que en ese caso se redirija al usuario a la página '/record/0/0'.
- Si $server es igual o mayor que la longitud del *array* de servidores provocará un error "out of index". Para evitarlo se redirigirá al usuario hacia la página '/record/0/0'.

Si no cumple ninguna de las condiciones anteriores, el siguiente paso es comprobar la variable $scanner. En la primera comprobación observamos si '$scanner' es menor que cero. En caso de serlo redirigimos a '/record/0/0'. Para la siguiente comprobación necesitamos el número de escáneres que se realizaron en ese servidor.
Para contar el número de escáneres realizados por el usuario utilizamos la función 'getScannersIds', pasándole como parámetros el alias del usuario y el Id del servidor al que pertenece, calculado mediante la función 'getServersbyKey', que recibe como parámetro el alias del usuario y el índice del servidor a seleccionar ('$serverKey').

```
$stmt = $dbConnection->prepare("SELECT servers.id_server FROM servers INNER
JOIN users ON users.id = servers.id_user WHERE users.alias = ?");
$stmt->bind_param("s", $alias);
$stmt->execute();
$stmt->bind_result($serverAlias);
array = array();
while ($stmt->fetch()) {
    $array[] = $serverAlias;
    }
$dbConnection->close();
$stmt->close();
return $array[$serverKey];
```

Una vez obtenido el Id del servidor en cuestión, podemos ejecutar la función 'getScannersIds'.

```
$stmt = $dbConnection->prepare("SELECT scanners.id_scanner FROM users
INNER JOIN servers ON servers.id_user=users.id INNER JOIN scanners ON
scanners.id_server=servers.id_server WHERE servers.id_server=? AND
users.alias=?");
$stmt->bind_param("ss", $id_server, $alias);
```

La consulta anterior devuelve un *array* con todos los Ids de los escáneres realizados por el usuario en ese servidor. Si la variable $scanner es mayor o igual a la longitud del *array* obtenido redirigimos hacia la página '/record/0/0'.

En caso de que los parámetros no cumplan ninguna de las condiciones anteriormente dichas se le enviará la página 'RecordBase.html.twig' con los parámetros '$server' y '$scanner' solicitados y el array con los alias de los servidores para construir la cabecera de la página, obtenidos anteriormente con la función 'getServers'.

Función ‘getInfo’

Esta función es la encargada de obtener los datos de los escáneres según los parámetros $server y $scanner. El primer paso es calcular los Ids del servidor y del escáner a solicitar. Para ello utilizamos las funciones ‘getServersbyKey’ y ‘getScannersIds’ ya utilizados en la función ‘index’.

Una vez tenemos los Ids del servidor y del escáner llamamos a la función ‘getScanners’ para obtener los resultados.

```
$stmt = $dbConnection->prepare("SELECT servers.rootUrl, scanners.currentDate, scanners.route, results.id_scanner, results.parameter, results.protocol, results.vType FROM users INNER JOIN servers ON users.id = servers.id_user INNER JOIN scanners ON scanners.id_server=servers.id_server INNER JOIN results ON results.id_scanner = scanners.id_scanner WHERE servers.id_server= ? AND users.alias=? AND results.id_scanner=?");
$stmt->bind_param("ssd", $serverId, $userAlias, $scannerKey);
$stmt->execute();
$stmt->bind_result($rootUrl, $date, $route, $id_scanner, $parameter, $protocol, $vType);
```

Una vez obtenidos los resultados, devolvemos al usuario un objeto JSON que contiene un *array* con todos los resultados del escáner solicitado. El *fron-end* los interpreta y los muestra por pantalla.

7. Integración sqlmap

En este capítulo vamos a tratar la integración de sqlmap a la aplicación. En el capítulo 6, en la función 'scanner' hicimos un inciso para llevar toda la explicación de la integración de sqlmap a este capítulo. Recordemos que existían tres posibles modos de consulta: escaneo automático, peticiones GET y peticiones POST. Cada una de ellas ejecutará un comando diferente [3].

Recordemos también lo dicho en el capítulo 2 sobre sqlmap. Es un programa en código abierto que funciona mediante consola de comandos. Es por ello que la primera decisión sobre la integración de sqlmap ha sido copiar el código completo de la aplicación en una carpeta. Concretamente en la carpeta 'web'.

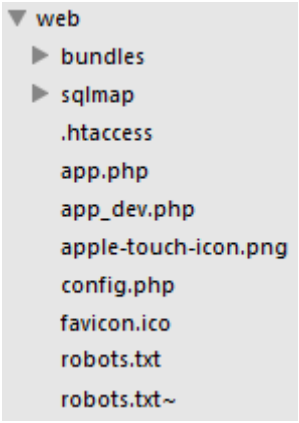

Imagen 11 - Carpeta Web

Con esta información vamos a proceder a explicar la integración de la herramienta a la aplicación.

7.1. Comandos

Como dijimos, sqlmap trabaja mediante línea de comandos. En PHP existe la posibilidad de ejecutar programas externos mediante línea de comandos utilizando la función 'shell_exec($comando)'. Donde '$comando' es el comando a ejecutar. Dentro de la carpeta de 'web/sqlmap' existe un ejecutable llamado 'sqlmap.py' que debemos ejecutar para correr sqlmap.

Cuando se pone en marcha un proyecto en Symfony se hace desde la carpeta 'web', ya que es desde donde se ejecutan los archivos 'app.php' o 'app_dev.php' (dependiendo del entorno elegido). Es por ello que para ejecutar el programa sqlmap almacenado en la carpeta 'web' desde Symfony debemos utilizar la siguiente función:

```
$resultado = shell_exec("./sqlmap/sqlmap.py" + $comandos);
```

La ejecución del comando anterior guardará en la variable '$resultado' un *string* con la salida por consola cuando termine su ejecución.

Una vez explicado el modo de ejecutar sqlmap dentro de la aplicación y cómo recoger los resultados es momento de explicar los diferentes comandos utilizados. En primer lugar, utilizamos ciertos comandos de forma sistemática en las tres formas de funcionamiento del programa. Estos comandos son:

- **'-u':** Indica la URL objetivo. Será una mezcla entre los datos 'url' y 'route' enviados por el usuario.

- **'--cookie':** Se utiliza en caso de ser necesaria la utilización de *cookies* para acceder al recurso objetivo. Primero observamos si el parámetro 'cookie' enviado por el usuario está vacío. En caso de no estarlo, añadimos el comando.
- **'--batch':** En ciertas ocasiones sqlmap solicita información en medio de la ejecución al usuario. Esto es un problema en nuestro caso ya que como no hay forma de responder a esas consultas la ejecución se para y no obtenemos los resultados solicitados. Con este parámetro forzamos a sqlmap a que se responda a sí mismo y siga con la ejecución del programa.
- **'--answers="Others=Y" ':** El problema de utilizar '—batch' es que en caso de que existan más parámetros a analizar, se auto-contestará de forma negativa evitando realizar el análisis de más parámetros. Al utilizar el parámetro 'answers' forzamos a 'batch' a responder lo que nosotros consideremos. En este caso, todas las consultas desde sqlmap que contengan la cadena "others" se contestarán positivamente.

```
[12:53:50] [INFO] testing 'Generic UNION query (NULL) - 1 to 20 columns'
[12:53:50] [INFO] automatically extending ranges for UNION query injection techn
ique tests as there is at least one other (potential) technique found
[12:53:52] [INFO] ORDER BY technique seems to be usable. This should reduce the
time needed to find the right number of query columns. Automatically extending t
he range for current UNION query injection technique test
[12:53:58] [INFO] target URL appears to have 1 column in query
[12:54:01] [WARNING] if UNION based SQL injection is not detected, please consid
er and/or try to force the back-end DBMS (e.g. '--dbms=mysql')
[12:54:01] [INFO] checking if the injection point on POST parameter 'alias' is a
 false positive
POST parameter 'alias' is vulnerable. Do you want to keep testing the others (if
 any)? [y/N]
```

Imagen 12 - Pregunta Sqlmap por línea de comandos

- **'--flush-session':** Sqlmap tiene la posibilidad de almacenar resultados de análisis anteriores y mostrarlos si un usuario los solicita de nuevo sin realizar ninguna comprobación. Para evita esto utilizamos el comando '–flush-session'.

Una vez vistos los comandos en común es momento de ver en qué se diferencian los tres modos de funcionamiento de la aplicación.

Modo 'Escaneo automático'

Este modo no recibe los datos 'param', 'paramValue', 'param1' o 'paramValue1' desde el usuario. En el controlador Main se comprueba si 'sForm' es igual a TRUE. De serlo, creamos el comando para este modo en la clase ClassMainFunctions mediante el siguiente código:

```
$comando = "./sqlmap/sqlmap.py -u ".$params['url']."/".$params['route'];
if(empty($params['cookie'])==false){
    $comando .= " --cookie=".$params['cookie'];
    }
$comando .= " --forms --batch --answers=\"Others=Y\" --flush-session;";
return $comando;
```

Este caso es el más simple y no tiene mayor complicación interpretar el código. Un ejemplo de una posible salida de esta función sería lo siguiente:

```
"./sqlmap/sqlmap.py -u http://www.tfmdlrtest.es/login.php --forms –batch --answers=”Others=Y” –flush-session;
```

Con este código, sqlmap obtiene los parámetros que se encuentran en esa ruta y ejecuta un análisis.

Modo protocolo POST

El modo parámetros con protocolo POST se reciben los parámetros ‘param’ y ‘paramValue’ y en caso de seleccionar la opción “Añadir nuevo parámetro” en la página Main, también se recibirán los parámetros ‘param1’ y ‘paramValue1’. El código para crear el comando es el siguiente:

```
$comando = "./sqlmap/sqlmap.py -u ".$params['url']."/".$params['route']." --data=\"".$params['param']."=".$params['paramValue'];
if(sizeof($params)==10){
    $comando .= "&".$params['param1']."=".$params['paramValue1'];
}
$comando .= "\"";

if(empty($params['cookie'])==false){
    $comando .= " --cookie=".$params['cookie'];
}
$comando .= " --batch --answers=\"Others=Y\" --flush-session;";
return $comando;
```

Es muy parecido al escaneo automático, salvo dos pequeñas diferencias. La primera, se le añade el comando ‘–data’ para introducir los datos recibidos en ‘param’ y ‘paramValue’. Acto seguido se comprueba si la longitud del *array* de los parámetros totales recibidos es igual a 10. Si es igual a 10 significa que el usuario ha pulsado el botón de “Añadir nuevo parámetro” y por tanto envía dos pares de parámetros. En ese caso se añaden los nuevos parámetros al comando. Un ejemplo de un análisis con protocolo POST y dos pares de parámetros puede ser:

```
"./sqlmap/sqlmap.py -u http://www.tfmdlrtest.es/login.php –data = ”alias = diego & password = 1234” –batch –answers = ”Others=Y” –flush-session;
```

Modo protocolo GET

Este modo es muy parecido al anterior. La única diferencia se encuentra en la forma de introducir los parámetros enviados por el usuario.

```
$comando = "./sqlmap/sqlmap.py -u ".$params['url']."/".$params['route']."?" .$params['param']." = ".$params['paramValue'];
if(sizeof($params)==10){
    $comando .= "&".$params['param1']."=".$params['paramValue1'];
}
if(empty($params['cookie'])==false){
    $comando .= " --cookie=".$params['cookie'];
}
$comando .= " --batch --answers=\"Others=Y\" --flush-session;";
return $comando;
```

Un ejemplo de un análisis con protocolo GET con *cookie* y un par de parámetros puede ser:

"./sqlmap/sqlmap.py -u http://www.tfmdlrtest.es/buscar.php?postId=2 –cookie = ”PHPSESSID = m12r6d0d7ocr1u5st4” –batch –answers = ”Others=Y” –flush-session;

7.2. Tratamiento de resultados

Como dijimos en el apartado anterior, los resultados se guardan en un *string* al finalizar la ejecución. Dentro de ese *string* se encontrará mucha información (tipo de pruebas realizadas, peticiones realizadas, etc.) que queda fuera de entendimiento de un usuario no experto. Es por ello que la información que vamos a tratar se tendrá que reducir lo máximo posible. Los parámetros elegidos para mostrar al usuario serán: el parámetro vulnerable, su protocolo y el tipo de vulnerabilidad SQL *Injection*.

La forma en que sqlmap muestra los resultados es sencilla ya que están todos entre dos cadenas ‘---‘.

```
POST parameter 'password' is vulnerable. Do you want to keep testing the others
(if any)? [y/N] y
sqlmap identified the following injection points with a total of 68 HTTP(s) requ
ests:
---
Parameter: password (POST)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: alias=diego&password=1234' AND 4749=4749 AND 'akeK'='akeK

Parameter: alias (POST)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: alias=diego' AND 3270=3270 AND 'exaH'='exaH&password=1234

    Type: AND/OR time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (SELECT)
    Payload: alias=diego' AND (SELECT * FROM (SELECT(SLEEP(5)))HVuM) AND 'CKrU'=
'CKrU&password=1234
---
```

Imagen 13 – Ejemplo de resultados en Sqlmap

El primer filtro que utilizaremos justo después de obtener los resultados será:

```
$sResultado=split('---',$resultado);
```

Si no encontramos ninguna cadena ‘---‘ significará que ha ocurrido un error en el análisis. Aplicamos un filtro para saber si los parámetros no son inyectables o se ha producido un error en los datos enviados. En esos casos responderemos al usuario con el mensaje correspondiente.

Si todo es correcto, el siguiente paso es interpretar los datos obtenidos después del filtrado. Para ello utilizamos la función ‘parseResults’ de la clase ClassMainFunctions. El código utilizado es:

```
$final=array();
$result=array();
```

```
$result=split('Parameter:',$escaner);
$parametro=0;
$protocolo=0;

for ($i=1; $i < sizeof($result); $i++) {

    $result2=explode("\n", $result[$i]);
    $vuelta=0;
    for ($j=0; $j < sizeof($result2)-1; $j++) {
            if($j==0){
                    $varParam=explode("(",$result2[$j]);
                    $parametro=$varParam[0];
                    $final[$i-1][$vuelta][0]=$parametro;
                    preg_match('#\((.*?)\)#', $result2[$j], $varProtocol);
                    $protocolo=$varProtocol[1];
                    $final[$i-1][$vuelta][1]=$protocolo;
            }else{
                    $varType=split('Type:',$result2[$j]);
                    if(sizeof($varType)>1){
                            if($vuelta==0){
                                    $final[$i-1][$vuelta][2]=$varType[1];
                                    $vuelta=$vuelta+1;
                            }else{
                                    $final[$i-1][$vuelta][0]=$parametro;
                                    $final[$i-1][$vuelta][1]=$protocolo;
                                    $final[$i-1][$vuelta][2]=$varType[1];
                                    $vuelta=$vuelta+1;

                            }

                    }

            }


    }
}
return $final;
```

Lo primero que realiza el código anterior es dividir los datos recibidos según la palabra “Parameter”. Con ello tendremos un *array* de *strings* que corresponderán a cada parámetro analizado. Dentro de cada posición del *array* vamos a realizar otro filtro para dividir el contenido por líneas. Cada posición del nuevo *array* es una línea diferente del texto original. En este punto tenemos un *array* bidimensional donde las “filas” son los textos de cada “Parameter” y las columnas son las líneas que tiene cada uno de ellos.

Con el *array* anterior lo primero que hacemos es coger el parámetro (mediante “$varParam=explode("(",$result2[$j]);”) y después el protocolo (mediante “preg_match('#\((.*?)\)#', $result2[$j], $varProtocol);”).

Una vez que obtenemos el protocolo y el parámetro vamos a recorrer el resto de líneas para obtener el tipo de vulnerabilidad. Como podemos observar en la imagen 13, cabe la posibilidad que dentro de un “Parameter” se encuentren dos tipos de vulnerabilidad.

Al final de la función nos quedará una matriz tridimensional [i][j][w] en donde almacenaremos todos los datos. Para los resultados mostrados en la imagen 13 tendremos la siguiente estructura:

```
i => 0
            j => 0
               w => 0, Valor =>  password
               w => 1, Valor => POST
               w => 2, Valor =>  boolean-based blind
            j => 1
               w => 0, Valor =>  password
               w => 1, Valor => POST
               w => 2, Valor =>  AND/OR time-based blind
i => 1
            j => 0
               w => 0, Valor =>  alias
               w => 1, Valor => POST
               w => 2, Valor =>  boolean-based blind
```

Imagen 14 - Estructura del array de salida

7.3. Guardar resultados

Una vez obtenido el *array* anterior es momento de almacenarlo en la base de datos. Para ello utilizamos la función ‘saveScanner’ de la clase DBClassMain. Esta función recibe el alias del servidor, la ruta del parámetro y el *array* del apartado anterior.

El primer paso es guardar la información en la tabla ‘scanners’ mediante la consulta:

```
$stmt = $dbConnection -> prepare("INSERT INTO scanners (route, currentDate, id_server) SELECT ?, NOW(), id_server FROM servers WHERE servers.serverAlias= ? ");
$stmt->bind_param("ss", $route, $serverAlias);
```

Una vez almacenados los datos en tabla ‘scanners’, cogemos el Id del escáner y lo utilizamos para guardar los resultados:

```
$last_id_inserted = $dbConnection->insert_id;
```

```
for ($i=0; $i < sizeof($result); $i++) {
    for ($j=0; $j < sizeof($result[$i]); $j++) {
$stmt2 = $dbConnection->prepare("INSERT INTO results (id_scanner, parameter,
protocol,vType) VALUES (?,?,?,?)");

$stmt2 -> bind_param("dsss", $last_id_inserted, $result[$i][$j][0], $result[$i][$j][1],
$result[$i][$j][2]);

$stmt2->execute();

    }
}
```

8. Web de pruebas y ejemplo de uso de la aplicación principal

8.1. Web de pruebas

Para comprobar el resultado de la aplicación anteriormente descrita he desarrollado una pequeña web de pruebas. Es una web muy sencilla, en la que no voy a entrar en detalle ya que solamente es para probar el funcionamiento de la web principal. Consta de tres páginas: 'login.php', 'index.php' y 'buscar.php'. La aplicación tendrá dos puntos vulnerables en los que nos vamos a centrar: formulario mediante POST en la página 'login.php' para dar acceso a la aplicación y consulta de información a través del protocolo GET en la página 'buscar.php'. Para ello se realizan las siguientes consultas vulnerables a *SQL Injection*:

Página 'login.php'

Se realiza la siguiente consulta a través de un formulario para acceder al sistema.

```
SELECT count(*) FROM users WHERE alias = '$user' AND password = '$password';
```

Donde '$user' y '$password' son parámetros enviados por el usuario. No se aplica ninguna medida de seguridad por lo que la consulta es completamente vulnerable. Como podemos observar, la consulta devuelve el número de columnas que coinciden con la condición impuesta en 'WHERE'. El resultado se examina y de ser mayor de 0 se redirige al usuario a la página Main. Como podemos observar, no mostramos ninguna información de la base de datos al usuario, por lo que la consulta será vulnerable a algún tipo de inyección tipo *Blind*.

Página 'buscar.php'

En la página 'buscar.php' se realiza la siguiente consulta para obtener por pantalla la información facilitada por el usuario mediante el protocolo GET.

```
SELECT post_id,mypost,date_time FROM posts INNER JOIN users ON users.id =
posts.id_user WHERE users.alias = '$alias' AND posts.post_id='$id';
```

El $id es enviado por el usuario y no se aplica ninguna protección. El resultado es mostrado por pantalla por lo que es probable que sea vulnerable a algún tipo de inyección SQL básica.

8.2. Ejemplo de uso de la aplicación principal

Para el ejemplo de uso vamos a realizar tres análisis diferentes. Cada uno de los análisis corresponderá con cada modo de funcionar de la aplicación principal. El primer análisis será utilizando el 'Escaneo automático' en la página 'login.php'. El segundo se realizará también sobre la página 'login.php' y sobre el mismo formulario (parámetros 'alias' y 'password') para poder ver la diferencia entre un escaneo automático y uno en donde introduzcamos valores correctos a los parámetros. Por último realizaremos un análisis en la página 'buscar.php' mediante el parámetro 'postId'.

Recordar que ambas páginas web están accesibles desde Internet desde donde se puede probar lo desarrollado o hacer las pruebas descritas en esta memoria. La URL de la aplicación principal es http://tfmdlrprod.es/ y la URL de la web de prueba es http://tfmdlrtest.es/ .

Antes de comenzar es recomendable instalar un proxy en el navegador para obtener el nombre exacto de los parámetros así como las *cookies* utilizadas en caso de estar en una zona con sesión abierta. Mi recomendación personal es Tamper Data en los navegadores Firefox. Es muy fácil de utilizar y obtener rodos los datos enviados al servidor.

Análisis de escaneo automático en la página 'login.php'

Para el primer análisis no necesitamos realizar ninguna intercepción de datos con Tamper Data ya que sqlmap se encarga de reconocer automáticamente las entradas de la ruta facilitada. En la imagen 15 podemos ver los datos enviados en este escaneo automático.

En la imagen 16 vemos el resultado del escaneo automático. Como podemos ver, el sistema sólo ha detectado el parámetro 'alias' como vulnerable, cuando en realidad, tanto 'alias' como 'password' son vulnerables. Esto es debido a que sqlmap no es capaz de detectar el comportamiento real de la aplicación al no tener un parámetro correcto.

Análisis con protocolo POST en la página 'login.php'

Para obtener el nombre de los parámetros que se envían en la página objetivo debemos interceptar una comunicación cliente/servidor mediante un proxy. En la imagen 17 vemos como he conseguido tal propósito utilizando Tamper Data. Los datos enviados en el análisis con parámetros han sido valores que existen en la base de datos (Imagen 18), y como podemos observar, el resultado del análisis (Imagen 19) cambia con respecto al escaneo automático.

En este caso ha detectado dos vulnerabilidades en el parámetro 'alias' por una en el parámetro 'password' debido a la autenticidad de la información enviada.

Análisis con protocolo GET en la página 'buscar.php'

La página 'buscar.php' se encuentra protegida, por lo que para acceder a ella el usuario se debe autentificar. Para esto es necesario obtener la *cookie* de sesión. En la imagen 20 muestro cómo he recogido la *cookie* con Tamper Data.

Esta vez no aparecen los parámetros en Tamper Data ya que las peticiones GET se recogen de la URL, justo después del carácter '?' y separados por el carácter '&'. En la imagen 21 vemos como el parámetro GET de la página 'buscar.php' se denomina 'postId'.

Los datos enviados (Imagen 22) son muy parecidos a los de un parámetro POST, salvo que en esta ocasión hemos necesitado añadir la *cookie* de sesión. Los resultados correspondientes se pueden ver en la imagen 23.

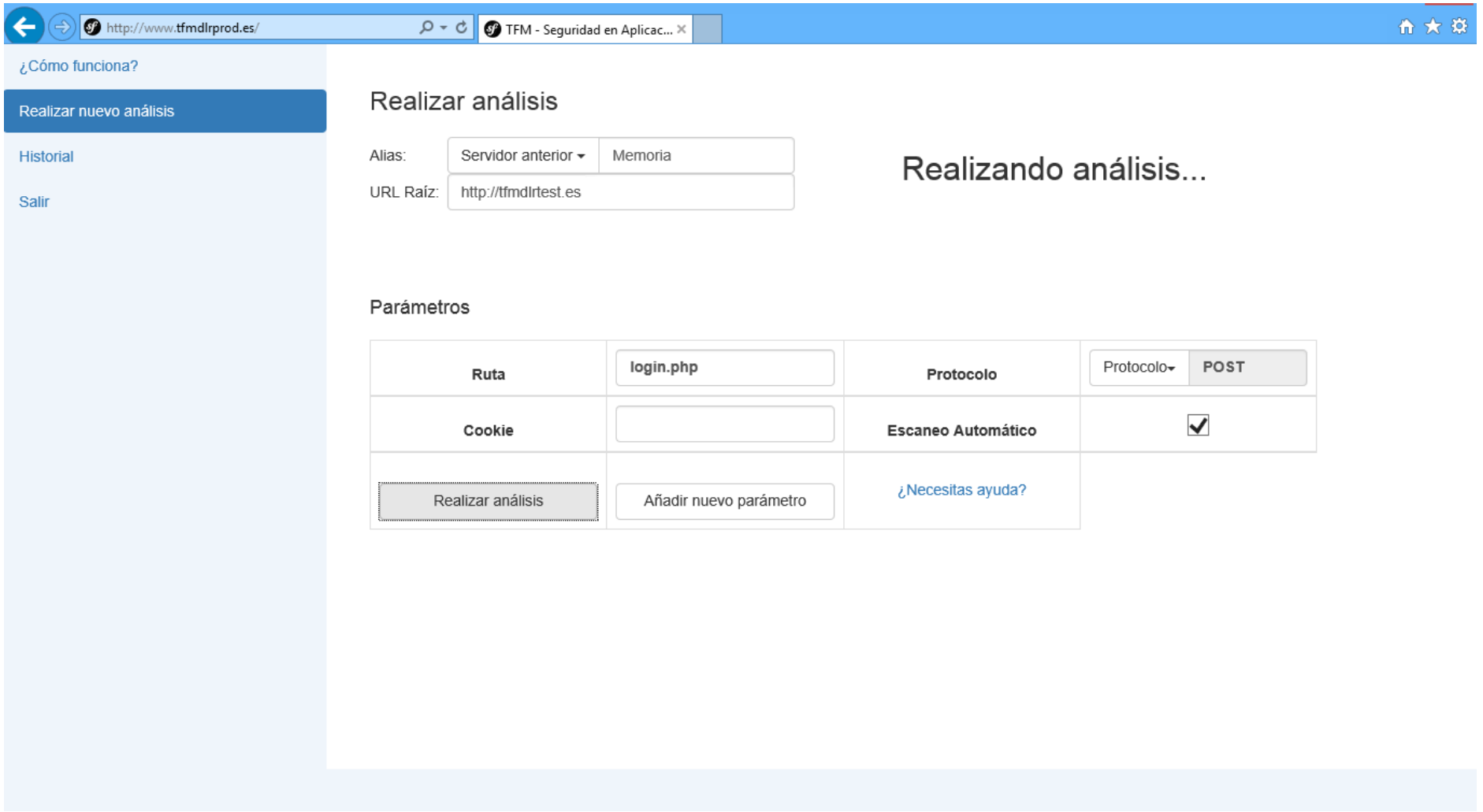

Imagen 15 - Escaneo Automático. Información enviada

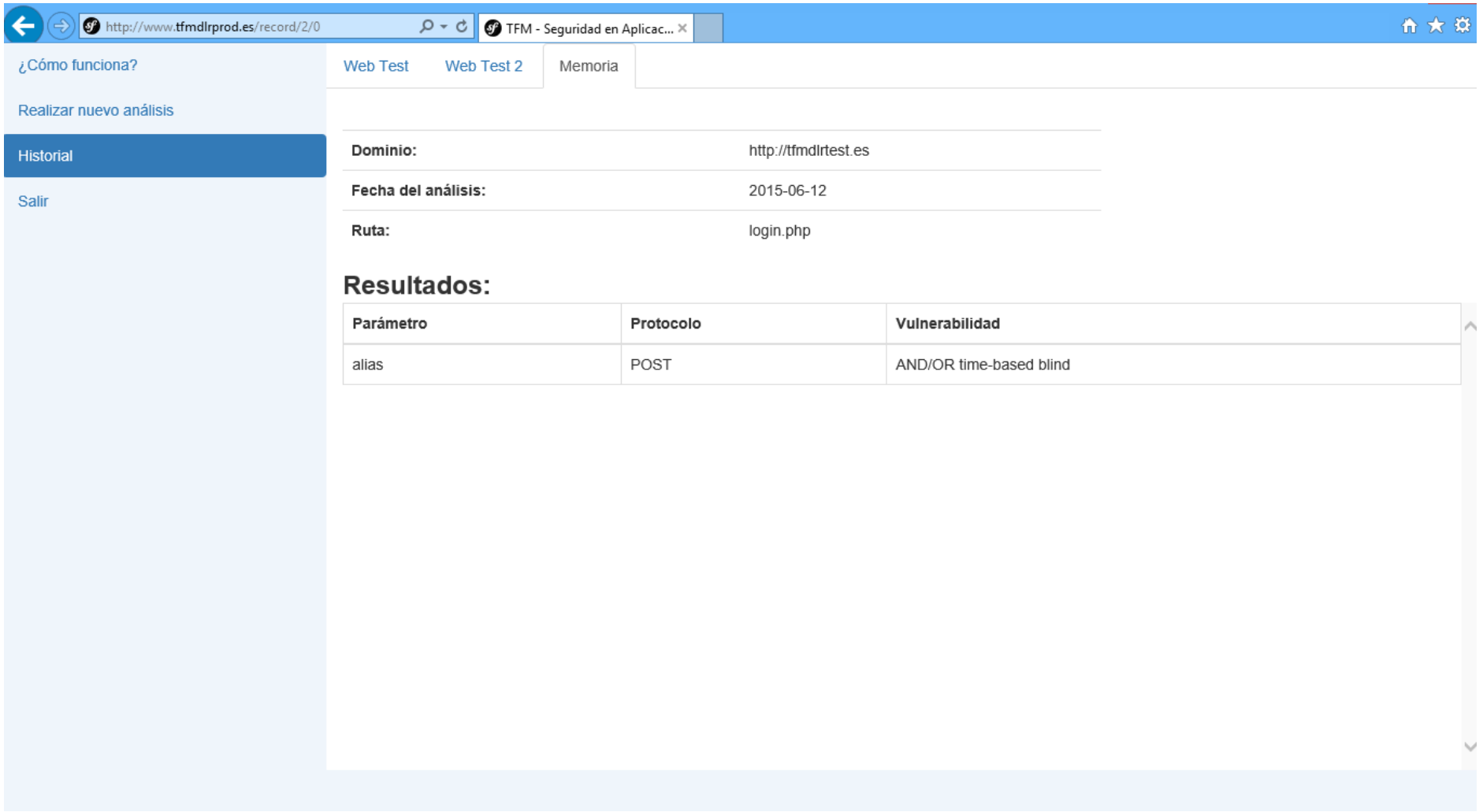

Imagen 16 – Escaneo Automático. Resultado

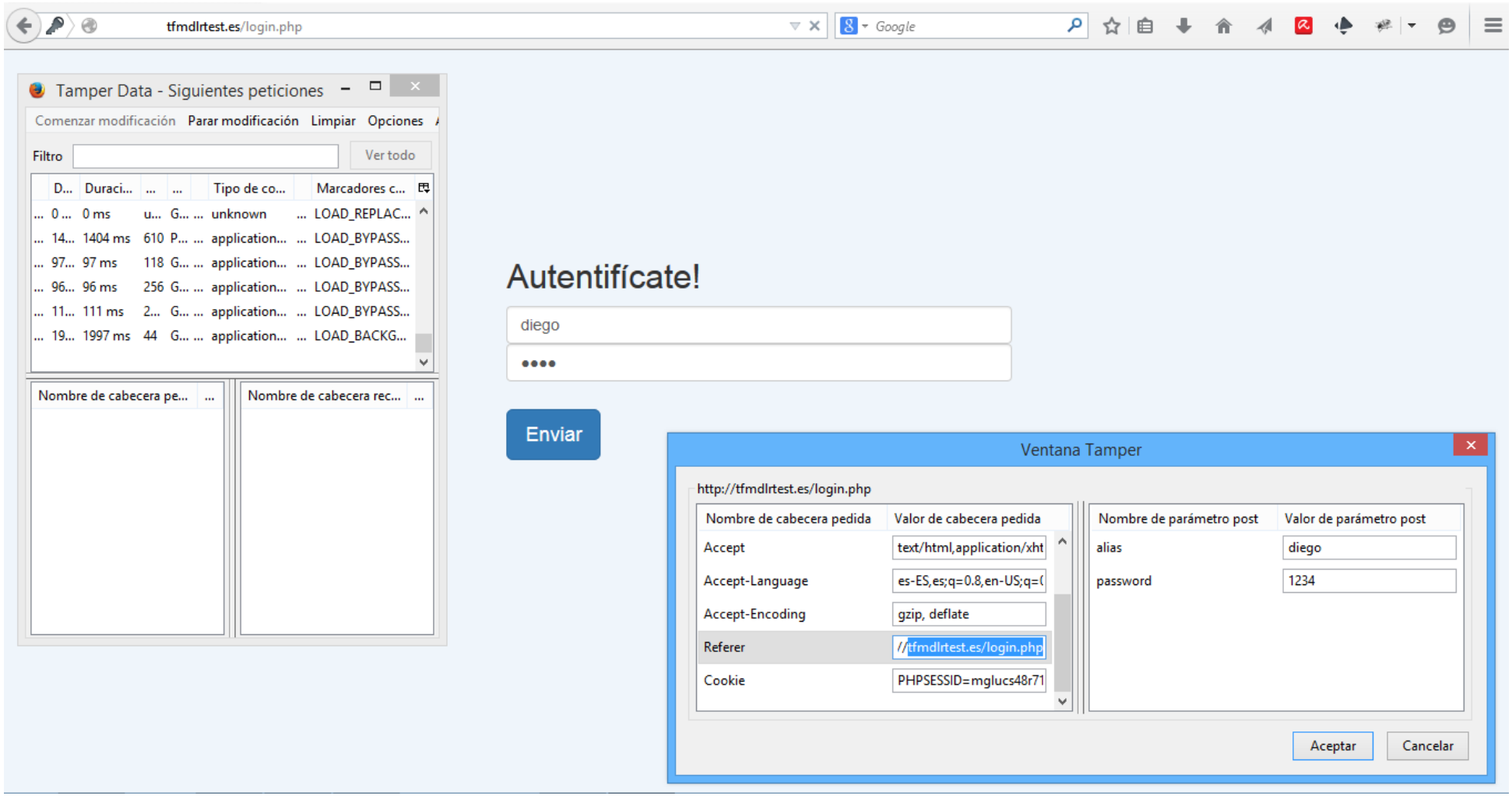

Imagen 17 - Uso de Tamper Data en comunicación POST

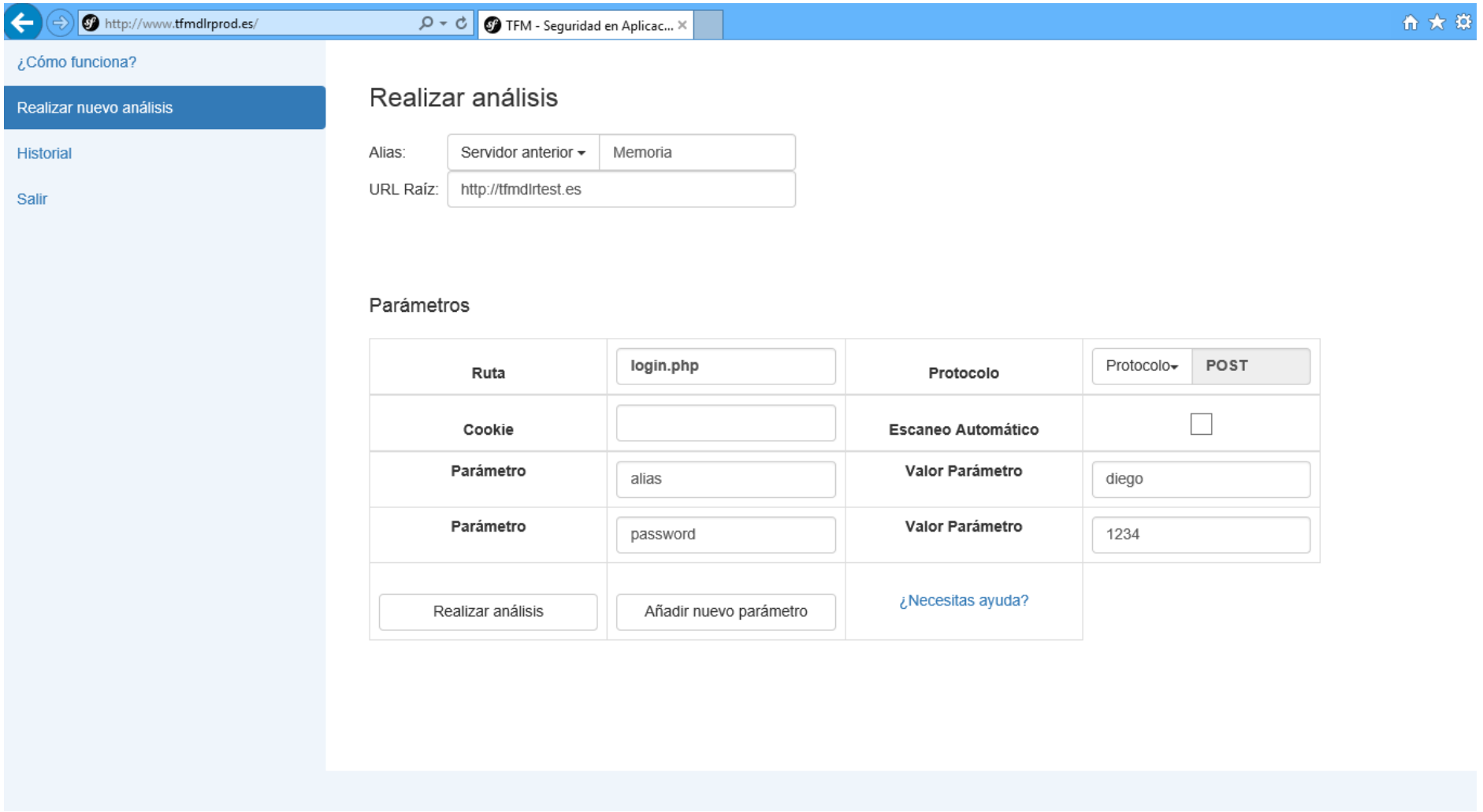

Imagen 18 - Análisis POST con parámetros. Información enviada

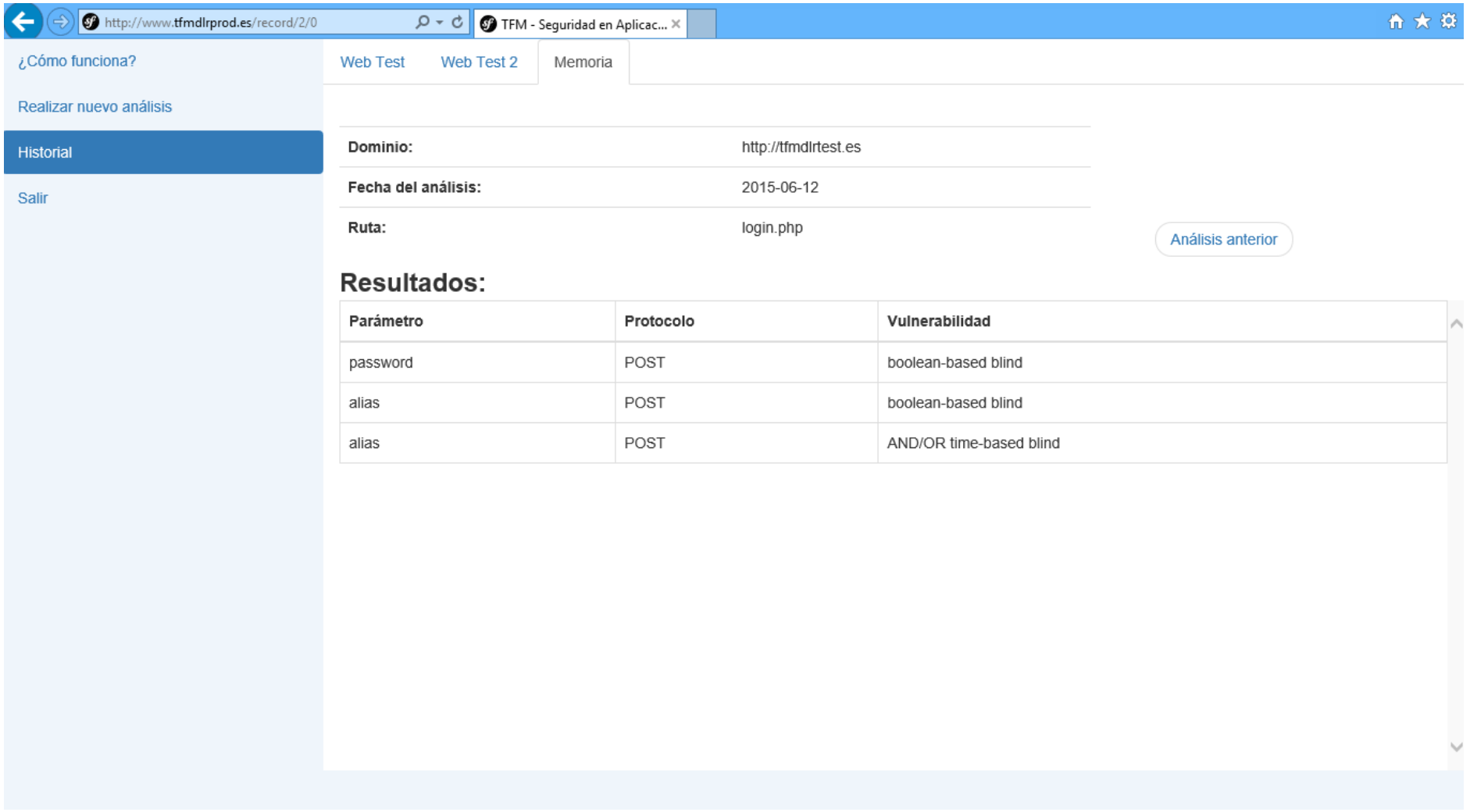

Imagen 19 - Análisis POST con parámetros. Resultado

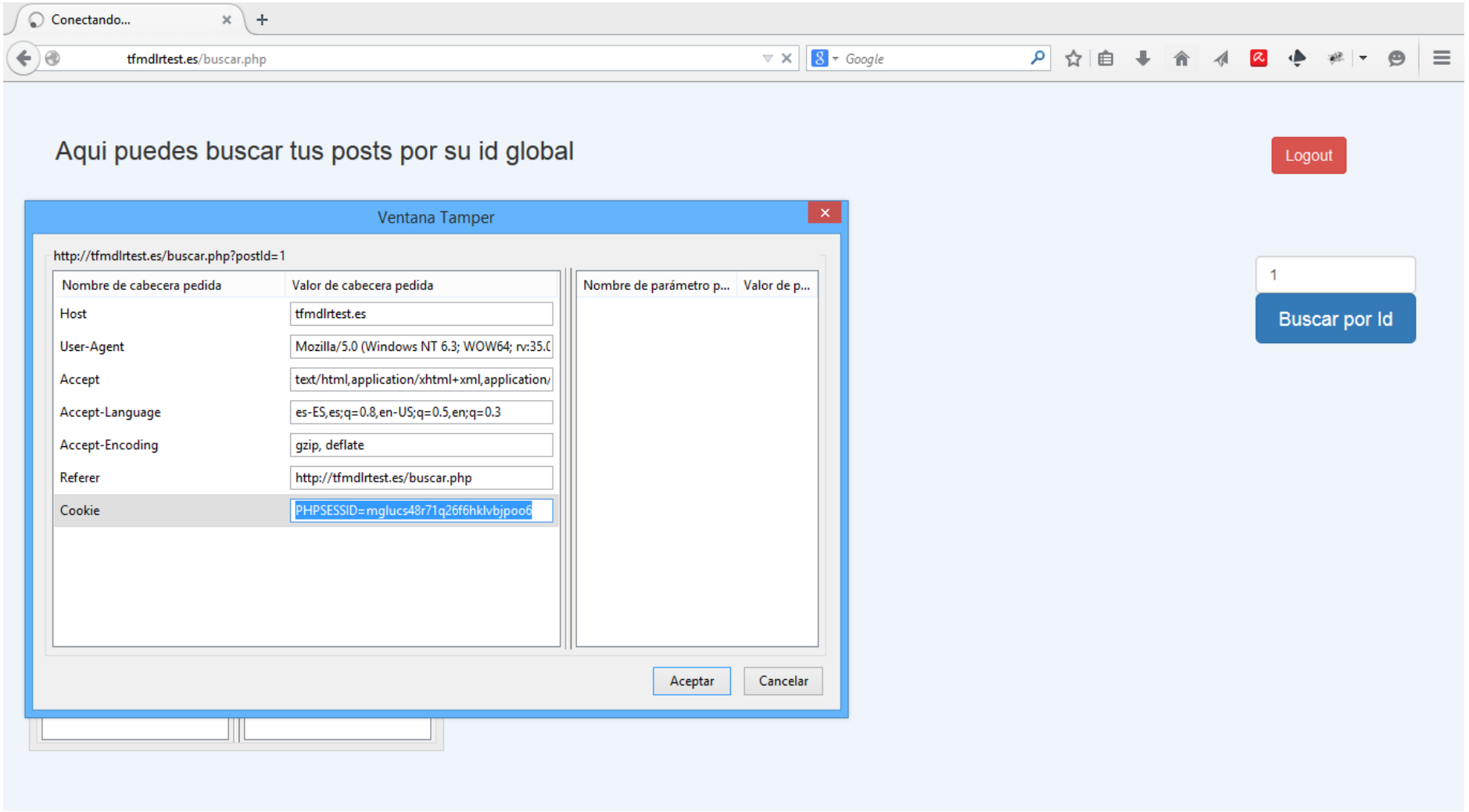

Imagen 20 - Obtención de cookie de sesión con Tamper Data

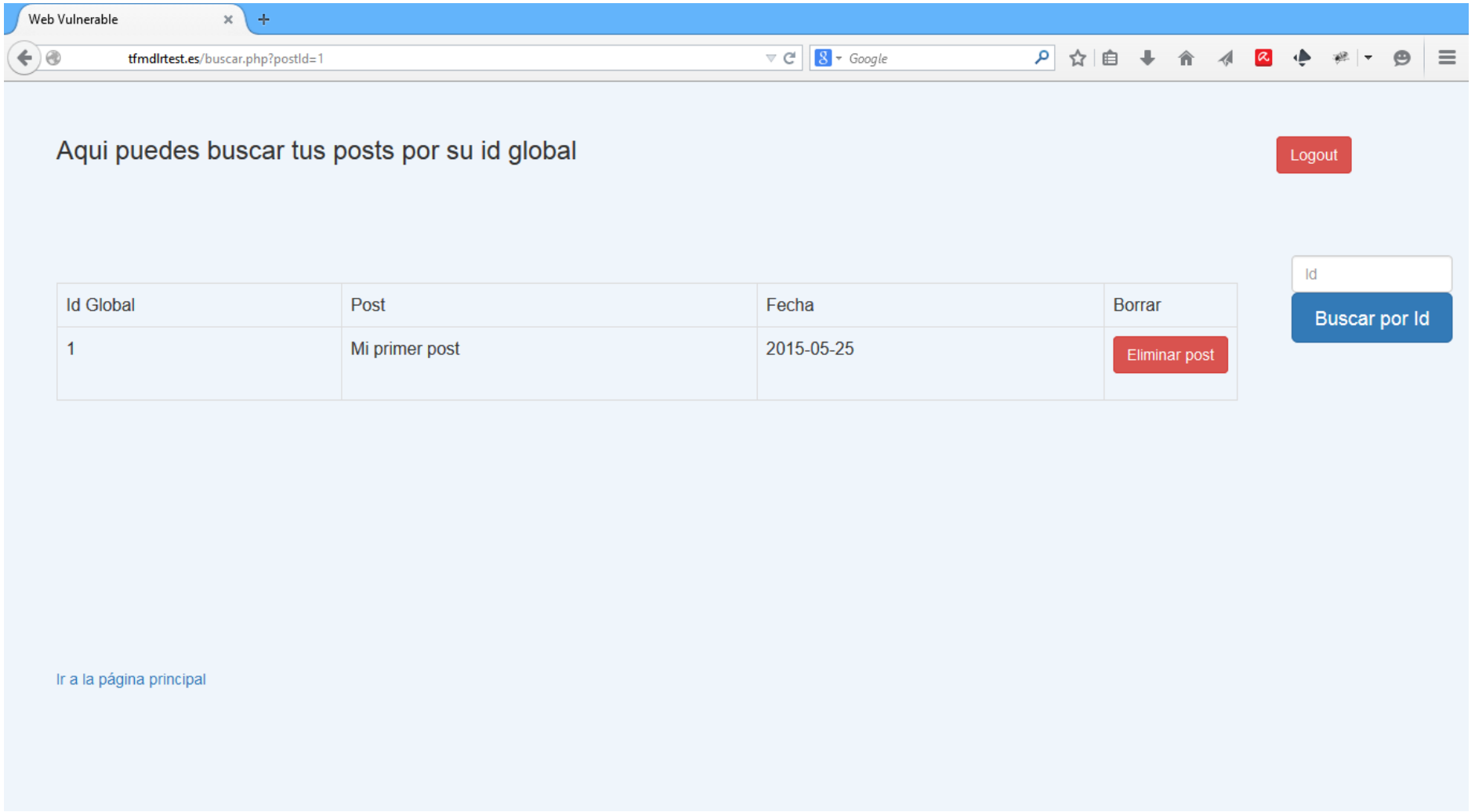

Imagen 21 - Parámetro GET de la página 'buscar.php'

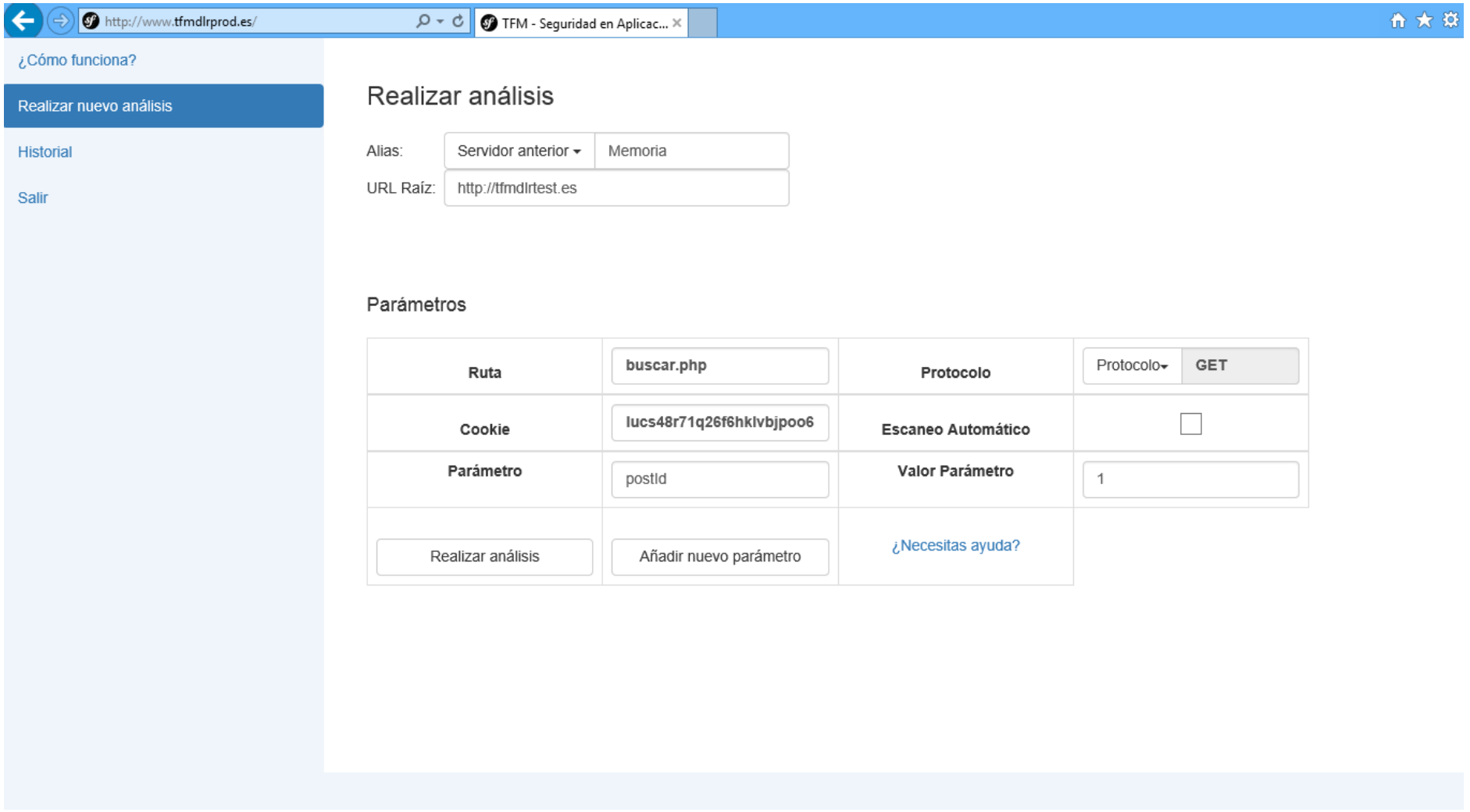

Imagen 22 - Análisis GET con parámetros. Información enviada

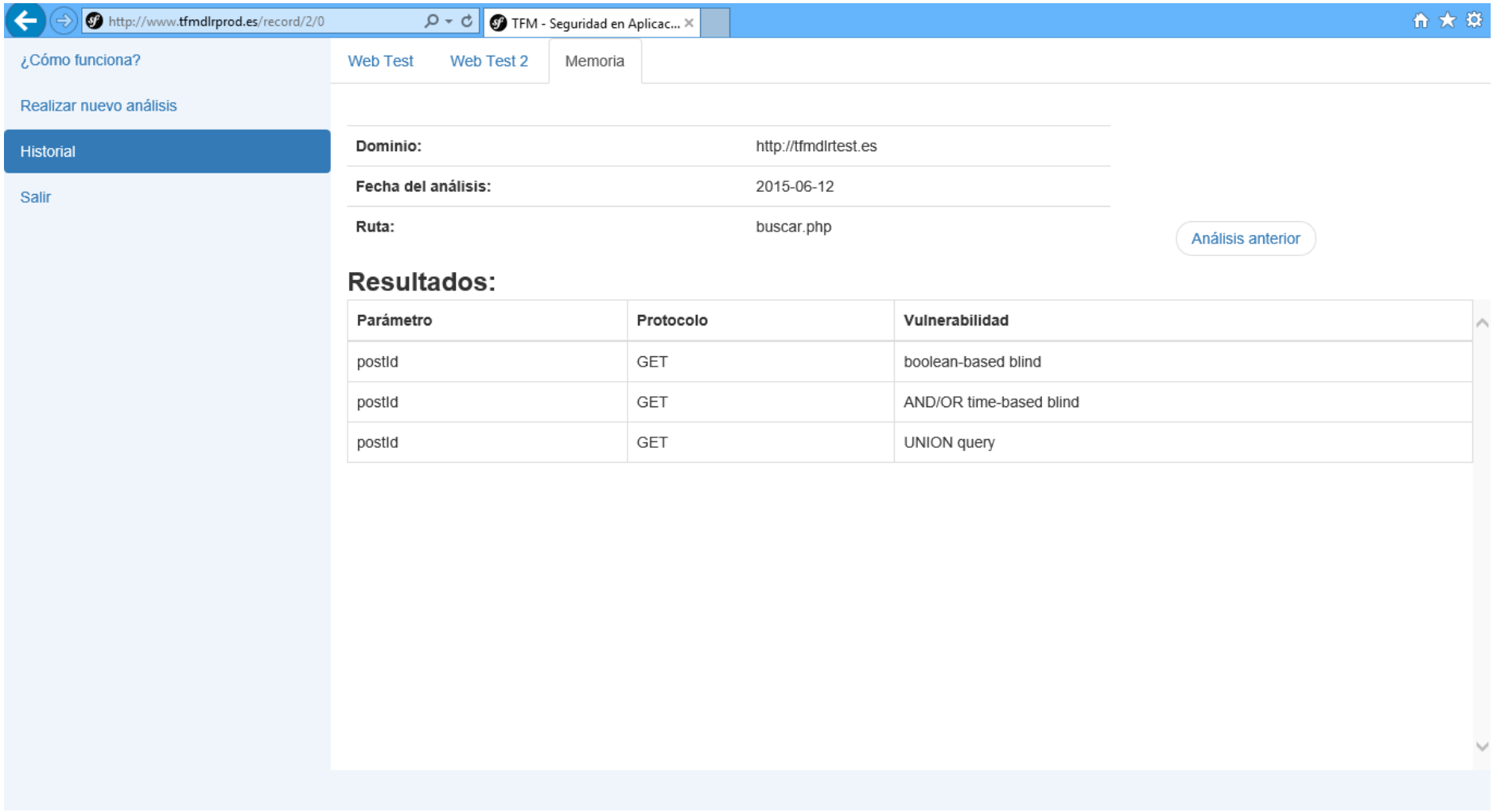

Imagen 23 - Análisis GET con parámetros. Resultados

9. Conclusiones y líneas futuras

Se han cumplido los objetivos marcados para el TFM, aunque creo que el resultado de alguno de ellos se podría mejorar. La idea de desarrollar una aplicación para no expertos ha sido complicada debido a la necesidad de la interacción de este con la aplicación (introducción de datos, elección del modo de funcionamiento, etc.). He intentado que la interfaz sea lo más amigable posible dentro de mis posibilidades, pero eso ha provocado una falta de información importante, sobre todo cuando se está aprendiendo a utilizar la aplicación.

Por otro lado, utilizar el comando 'shell_exec' tiene un gran inconveniente. El servidor se bloquea hasta que termina de ejecutar el comando. El proceso de análisis puede ser largo, en algunos casos llegando a durar más de un minuto. Esto provoca una inactividad en la aplicación bastante grande mientras se realiza un análisis, dando la sensación en algunos momentos de que ha ocurrido un error en la aplicación. Este problema es de difícil solución con las herramientas utilizadas.

En líneas futuras se podría replantear el diseño de la web Main y adaptarla mejor a usuarios no expertos con mensajes de ayuda dinámicos que no influyan en la navegación cuando el usuario ya conozca la aplicación. La opción de utilizar *pop-ups* podría resultar interesante para mejorar la experiencia de aprendizaje de la aplicación.

Otro aspecto interesante sería realizar una nueva página capaz de resumir todos los análisis realizados de un servidor y mostrarlos al usuario de manera gráfica. Esto ayudaría a llevar un mejor control de los servidores analizados.

10. Bibliografía

1. OWASP - Testing Guide v4: https://www.owasp.org/index.php/OWASP_Testing_Project
2. SQLMAP: http://sqlmap.org/
3. SQLMAP - Usage: https://github.com/sqlmapproject/sqlmap/wiki/Usage
4. SQLMAP – Techniques: https://github.com/sqlmapproject/sqlmap/wiki/Techniques
5. OWASP - TOP 10 2013 – Injection: https://www.owasp.org/index.php/Top_10_2013-A1-Injection
6. OWASP - Testing for SQL Injection: https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OTG-INPVAL-005)
7. OWASP – SQL Injection Prevention Cheat Sheet: https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
8. Symfony - Documentation: http://symfony.com/doc/current/index.html
9. jQuery – Api Documentation: https://api.jquery.com/
10. Bootstrap - Components: http://getbootstrap.com/components/
11. Twig: http://twig.sensiolabs.org/