Seguridad en aplicaciones Web

Memoria del proyecto de fin del Máster interuniversitario en Seguridad de las Tecnologías de la Información y de las Comunicaciones realizado por *Ricardo Roda Bravo* y dirigido por *Jordi Duch Gavaldà* y *Agustí Solanas Gómez*

Barcelona, 10 de Enero de 2014

Título: Seguridad en aplicaciones web
Autor: Ricardo Roda Bravo
Directores: Jordi Duch Gavaldà y Agustí Solanas Gómez
Fecha: 10 de Enero de 2014

Resumen

Este proyecto estudia la seguridad en aplicaciones centrándose en las vulnerabilidades de tipo inyección SQL y tiene como objetivo el desarrollo de una aplicación que realice tests a las páginas deseadas para comprobar que estas son atacables haciendo uso de esta vulnerabilidad

Encontraremos una primera parte con un pequeño resumen sobre esta vulnerabilidad y los diferentes tipos que existen de esta.

En la segunda parte veremos el desarrollo de la aplicación web y los resultados obtenidos.

Title: Security in web applications
Author: Ricardo Roda Bravo
Directors: Jordi Duch Gavaldà and Agustí Solanas Gómez
Date: 10th January 2014

Overview

This project studies security in web applications focusing on sql injection vulnerabilities and aims to develop an application that makes tests to web applications to check if they are exploitable using this kind of vulnerability.

In the first part we will find a brief abstract about this kind of vulnerability and the different types that we can find.

In the second part is where we can find the development of the web application and the results obtained.

Índice

Capítulo 1 – Introducción

1.1 - Justificación del proyecto

Una inyección SQL es una vulnerabilidad informática basada en introducir una sentencia SQL maliciosa aprovechando un fallo de seguridad en una página web, normalmente en un formulario.

Las consecuencias de sufrir un ataque por inyección SQL pueden ser devastadoras para un servicio. Se pueden sufrir desde modificación, eliminación y inserción de datos hasta filtración de datos personales.

Viendo lo graves que son las consecuencias de un ataque por inyección SQL no resulta difícil entender porqué la *Open Web Application Security Project (OWASP)* [1] la considera desde 2007 hasta la actualidad como una de las diez vulnerabilidades mas críticas, siendo en 2013 la número uno de la lista [2].

La idea de este proyecto es ofrecer una web sencilla donde poder escanear una página para comprobar que no es vulnerable a inyecciones SQL y ofrecer una herramienta sencilla para que el desarrollador sea consciente de esta vulnerabilidad y pueda poner remedio.

En ningún caso se pretende crear una herramienta para facilitar ataques a páginas que no sean propiedad del usuario que hace el test.

1.2 - Punto de partida

Existen muchas herramientas especializadas en escaneo de páginas web para detectar campos inyectables. Muchas de ellas requieren una instalación y/o uso complejos o son especializadas en DBMS específicos (*Data Base Management Systems*).

La idea es crear una web lo mas simple posible, que no requiera gran interacción con el usuario para que, hasta el desarrollador mas inexperto, pueda testear su aplicación, pero también dar la opción a usuarios mas avanzados poder probar de manera mas profunda la seguridad de su web contra este tipo de vulnerabilidad.

Esta web mostraría interactivamente los resultados que se obtienen al ejecutar, en segundo plano, una aplicación dedicada a la detección automática de campos inyectables en formularios vulnerables a inyecciones SQL. En este caso la aplicación elegida es **SQLMAP** [3].

La elección de *sqlmap* respecto a otras aplicaciones con la misma finalidad como podrían ser *wapiti* [4], *zed attack proxy* [5] o *fatcat* [6] es la potencia y versatilidad respecto a las demás ofreciendo gran cantidad de opciones siendo relativamente sencilla de usar.

Además, esta no requiere de ningún tipo de instalación de proxy, servidor web o incluso de entorno gráfico.

1.3 - Objetivos

El objetivo que se intentará conseguir es desarrollar una web capaz de realizar escaneos de URLs para comprobar si estas son vulnerables a inyecciones SQL.

La idea es ser capaz de hacerla lo mas simple posible para que un escaneo pueda ser ejecutado sin necesidad de tener grandes conocimientos en la materia.

Aún así no se quiere limitar esta web a escaneos simples y se pretende dar las máximas opciones posibles a aquellos usuarios mas avanzados que busquen escanear en mayor profundidad su aplicación.

Probablemente no será posible implementar todas las opciones que ofrece *sqlmap* ya que algunas de ellas seria demasiado complejo de realizar en un *frontend* web como es el caso de esta aplicación, otras porque simplemente no dará tiempo.

Aún así se intentará desarrollar primero las opciones mas básicas y poco a poco ir añadiendo aquellas que se consideren mas útiles.

Evidentemente el objetivo básico es poder hacer escaneos de URL automatizados pero también que se pueda añadir en la URL parámetros de formularios con envío por GET.

La siguiente opción prioritaria a implementar es añadir escaneo de parámetros de formularios con envío por POST.

Luego se irán implementando nuevas opciones interesantes, como poder elegir el tipo de retorno de datos en caso de poder hacer inyecciones SQL (retorno del usuario actual del *dbms*, retorno de tablas, de contenido, etc), poder especificar tanto el nivel como el riesgo de los tests que ejecute el *sqlmap*, o poder hacer uso de las diferentes optimizaciones que incorpora esta aplicación para intentar reducir el tiempo de ejecución de los testeos.

En caso de disponer de mas tiempo se intentará ir añadiendo progresivamente mas opciones.

Otro objetivo es conseguir que los resultados que se obtengan de la ejecución del *sqlmap* sean presentados al

usuario de manera inmediata, tal y como lo vería si estuviera ejecutándolo él mismo delante de un terminal. De aquí viene uno de los requerimientos de la aplicación de usar *ajax,* para poder hacer esto posible y mostrar al usuario estos resultados de una manera atractiva.

Por último otro objetivo que se buscará cumplir es permitir al usuario descargarse el resultado del test obtenido para su posterior estudio y no ser necesario tener que re-ejecutar el test de nuevo.

1.4 - Enfoque y método seguido

Se intentará poner especial hincapié en la seguridad de la web. No nos centraremos en hacer el servidor en si seguro (sistema operativo correctamente configurado, detectores de intrusiones, *firewall*, etc), aun así se ha prestado un mínimo de atención a configurar de manera correcta el servidor *tomcat* en el cual se ejecuta la aplicación.

La tecnología usada para desarrollar esta web ha sido java [7] (jsp con *javaserver pages standard tag library* (jstl) y *Expression Language* (EL)) y *jquery* [8], el cual usaremos, entre otras cosas, para usar la técnica de Javascript asíncrono y XML (o en su acrónimo en inglés, AJAX), tanto en la presentación de los datos como en la muestra del formulario.

El patrón de diseño elegido es el clásico *modelo-vista-controlador* (MVC).

En esta aplicación el modelo son dos clases de java: una dedicada a gestionar los parámetros obtenidos del usuario (*SQLMapParser.java*) y otra clase dedicada a la ejecución de *sqlmap* y retornar los datos obtenidos a la vista para que los muestre (*SQLMapRunner.java*). En esta parte es donde comprobamos que los datos del usuario son correctos.

La vista es donde usamos *jsp* y *javascript* (*jquery ajax*) para mostrar el formulario con las diferentes opciones y donde mostramos los datos obtenidos mientras realizamos el test.

El controlador es un *servlet* de *java* (*getURL)* que obtiene los datos del formulario obtenido en la vista y se los enviamos al modelo para que los gestione.

1.5 Planificación

El día 26 de septiembre de 2013 se inicia el proyecto.

Usaremos una semana para buscar información sobre inyecciones y diferentes herramientas, así como empezar a plantearse que tecnologías usar y en que entorno empezar el desarrollo.

Probablemente será una tarea que se irá retocando y perfilando a medida que se avance en el proyecto, pero en principio se dedicará dos semanas a decidir la estructura de la aplicación y los componentes que la formarán. A priori las tecnologías a usar y el patrón de diseño están claros y no variaran: *java* + *javascript* con *jquery* (quizás con el tiempo se decida usar algún *plugin* de *jquery* para temas de personalización, pero cosas menores) y *css* para el estilo. El patrón será el clásico MVC (Modelo-Vista-Controlador).

Una vez decididas las tecnologías y tener claro la temática a desarrollar dedicaremos unos 10 días a montar y securizar el entorno de trabajo y el servidor en que se ejecutará la aplicación.

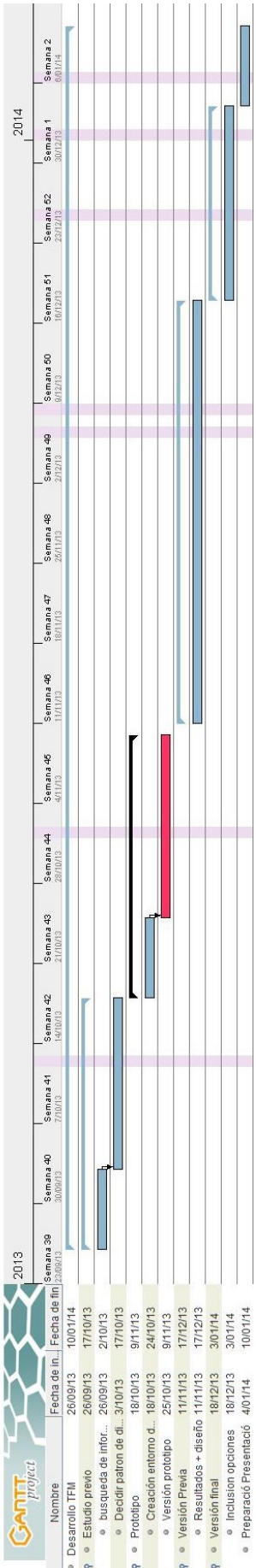
El desarrollo se prolongará hasta la semana final del proyecto si contamos la parte de depuración y mejoras, pero se le dedicará 15 días al desarrollo de una versión prototipo sin ningún tipo de estilo y con mínimas opciones. Dado que por los objetivos fijados hemos de usar una aplicación que ha de ser ejecutada en terminal por el sistema miraremos con especial cuidado el tema de la seguridad en su ejecución desde la web.

Una vez la ejecución del *sqlmap* se desarrolle correctamente pasaremos a desarrollar una versión web que muestre resultados y si vemos que podemos hacerla fácilmente ya iniciaremos también el proceso de ir añadiendo partes de carga en *ajax*. Iniciaremos el desarrollo del estilo básico. Le dedicaremos dos meses.

Llegados a este punto el desarrollo de la web tendría que estar suficientemente avanzado y estructurado como para poder añadir mas opciones de ejecución de manera relativamente rápida. Le dedicaremos unos 15 días a añadir todas las opciones que nos hemos planteado inicialmente en los objetivos. En caso de necesitar menos tiempo podríamos dedicarnos a añadir mas opciones de las planteadas inicialmente.

La memoria se irá desarrollando a medida que vaya avanzando el proyecto y la presentación final en la última semana.

Pasemos a mostrar el diagrama de Gantt de la planificación aquí explicada.


GANTT project
2013
2014
Nombre
Fecha de in...
Fecha de fin
Desarrollo TFM 26/09/13 10/01/14
Estudio previo 26/09/13 17/10/13
busqueda de infor... 26/09/13 2/10/13
Decidir patron de di... 3/10/13 17/10/13
Prototipo 18/10/13 9/11/13
Creación entorno d... 18/10/13 24/10/13
Versión prototipo 25/10/13 9/11/13
Versión Previa 11/11/13 17/12/13
Resultados + diseño 11/11/13 17/12/13
Versión final 18/12/13 3/01/14
Inclusion opciones 18/12/13 3/01/14
Preparació Presentació 4/01/14 10/01/14
Semana 39 23/09/13
Semana 40 30/09/13
Semana 41 7/10/13
Semana 42 14/10/13
Semana 43 21/10/13
Semana 44 28/10/13
Semana 45 4/11/13
Semana 46 11/11/13
Semana 47 18/11/13
Semana 48 25/11/13
Semana 49 2/12/13
Semana 50 9/12/13
Semana 51 16/12/13
Semana 52 23/12/13
Semana 1 30/12/13
Semana 2 6/01/14

1.6 - Organización de la memoria

Esta memoria se encuentra organizada en seis partes diferenciadas.

En esta primera parte encontramos la introducción, que es donde hacemos una justificación de este proyecto, los objetivos y motivaciones que me han llevado a desarrollarlo así como el enfoque y el punto de partida al iniciar este proyecto y la planificación.

En la segunda parte, que podríamos considerarla como una introducción de carácter más bien técnico, tenemos una explicación de que son la inyecciones SQL y que diferentes tipos de estas existen.

En la tercera parte es donde podremos encontrar una descripción de la aplicación: desde diagramas de flujo hasta un análisis de requisitos.

En la cuarta parte explicamos la estructura básica de la aplicación así como las partes que la forman.

La quinta parte es donde podremos encontrar el seguimiento de la implantación de la aplicación: entorno utilizado y resultados conseguidos, etc.

Para finalizar, en la última parte es donde encontraremos las conclusiones a les que se ha llegado a partir del trabajo realizado y que trabajo futuro se podría realizar para mejorar la aplicación.

Capítulo 2 – Inyección SQL, definición y tipos

2.1 – Definición

Una inyección SQL es un método de inclusión de código no deseado aprovechándose de una vulnerabilidad en la aplicación resultando en una ejecución de sentencias SQL no deseadas o previstas.

La manera mas habitual en que se suele poder explotar este método es en formularios web, todo y que no es la única manera. Cualquier aplicación que haga consultas a base de datos dependiendo de un valor introducido por el usuario es sensible a sufrir este tipo de ataques si no se usan los métodos necesarios de control para evitarlos.

2.2 - Tipos de Inyección SQL

Existen diferentes tipos de inyección SQL dependiendo del método que se aprovecha para poder realizar el ataque. Podríamos hacer la clasificación en cinco tipos diferentes:

2.2.1 - Filtración incorrecta de *strings*

Son las inyecciones mas simples que se pueden realizar dado que indican que no hay ningún tipo de control ni prevención a este tipo de vulnerabilidades.

La manera mas simple (que no mas segura) de realizar un formulario web que necesita hacer consultas a base de datos es crear una variable con una consulta SQL donde el campo *WHERE* queda a expensas del valor que recuperamos del formulario. Normalmente en el condicional deberemos abrir un *tick* ('), poner el valor recibido del formulario y cerrar el *tick*. Una vez tenemos la consulta completa la podemos ejecutar. Un ejemplo seria

```
$query="SELECT pass FROM users WHERE username = '" . $_POST['username'] .
"';";
$result = mysql_query($query);
```

Si las instrucciones se ejecutan tal como se ve, sin hacer ningún tipo de control, es fácil ver que un atacante podría, en el propio formulario, cerrar el *tick* abierto y forzar a la consulta retornar un valor cierto (*true*).

Por ejemplo, si en el formulario del ejemplo anterior, el atacante en vez de poner su usuario introdujera lo siguiente:

```
' or 'x' = 'x
```

Se dará que en la consulta que tenemos programada quedaría, substituyendo el parámetro *$_POST['username']* por el valor que ha puesto el atacante:

```
"SELECT pass FROM users WHERE username = '' or 'x' = 'x';";
```

Haciendo un simple filtrado al *string* de entrada por el usuario, escapando caracteres se podría evitar este ataque.

2.2.2 - Control incorrecto de tipo de datos

Este tipo de inyección aprovecha que no se comprueba que el valor introducido realmente es el que el programador esperaba.

El caso mas típico es esperar que un usuario nos introduzca un valor numérico y, si no se hace ningún tipo de comprobación, un atacante podría introducir una cadena de caracteres maliciosa. Por ejemplo en la siguiente consulta:

```
$query="SELECT * FROM users WHERE id = " . $_POST['id'] . ";";
```

Si no se hace una comprobación de que el valor *id* del formulario es realmente un número un atacante podría introducir lo siguiente:

```
1; DROP TABLE users
```

Quedando la consulta que se ejecutará finalmente será la siguiente:

```
"SELECT * FROM users WHERE id = 1; DROP TABLE users;";
```

Para evitarlo simplemente podríamos usar la función *is_numeric* para comprobar realmente que se está introduciendo un número.

2.2.3 - Evasión de control por signaturas

Ciertas inyecciones SQL pueden ser evitadas usando diferentes sistemas de prevención basados en el uso de signaturas, como podría ser los sistemas de detección de intrusiones.

Existen diferentes tipos dependiendo de los diferentes métodos que se pueden usar para evitar estos tipos de detección. Los métodos mas comunes son: usar diferentes tipos de codificación, alternativas a espacios y inserción de patrones de *string* arbitrarios.

El método de uso de codificación diferente se basa en no usar la codificación estándar para evitar las comprobaciones de ciertos detectores de intrusiones. Por ejemplo la sentencia

```
NULL OR 1=1
```

Podría evitarse el control de signaturas usando por ejemplo una codificación URL

```
NULL+OR+1%3D1%
```

El otro método mencionado de usar alternativas a los espacios simplemente se basa en evitar usar espacios por otros caracteres que una sentencia aceptaría como separador de campos pero un detector basado en signaturas no capturaría como sospechoso. Por ejemplo podemos sustituir espacios por tabuladores o saltos de linea.

En la sentencia del ejemplo anterior, usando esta técnica quedaría de la siguiente manera:

```
NULL
OR
1
=
1
```

El último método mencionado sería parecido al anterior. Se basa en añadir caracteres aceptados por una consulta pero que servirían para hacer que ciertos detectores de intrusiones los pasaran por buenos. Por ejemplo, se pueden introducir comentarios estilo C/C++ que una consulta acepta por buenos y ignora en la ejecución. En el ejemplo anterior sería:

```
NULL/**/OR/**/1/**/=/**/1/*evitar detectores*/
```

2.2.4 - Bypass de filtros

Estas técnicas se basan en intentar aprovechar el comportamiento de diferentes filtros (*addslashes*, *magic_quotes_gpc*, etc) que existen para intentar evitar este tipo de vulnerabilidad. Existen varios modos de intentar saltarse estas filtraciones.

Un ejemplo sencillo para explicar este método es el uso de listas negras de expresiones que se eliminan al estar presentes en el contenido de los formularios. Normalmente tendremos en esta lista expresiones del estilo *OR,UNION, SELECT, WHERE,* etc. Cuando recibimos el formulario escaneamos el contenido de este usando la función *str_replace* donde vamos remplazando, del *string* introducido en el formulario, todas estas expresiones de la lista negra por un *string* vacío para eliminarlas.

El problema que se produce es que *str_replace* solo hace la sustitución del *string* dado sin usar recursividad en el *string* generado. Por ejemplo si tenemos el *string* "eepp" y substituimos "ep" por un *string* vació nos quedaría lo siguiente:

```
str_replace("eepp","ep",""); → ep
```

Nunca nos quedará un *string* vació por mucho que la palabra resultante también nos diría la lógica que debería ser remplazada.

Un atacante que conozca este comportamiento lo tendría relativamente fácil para saltarse este filtro. Simplemente hay que duplicar el contenido de las palabras que sospechamos que están dentro de la lista negra. Por ejemplo si queremos inyectar la siguiente sentencia SQL:

```
SELECT password FROM users WHERE name LIKE 'Admin%'
```

Simplemente tendría que introducirlo en el formulario de la siguiente manera:

```
SSELECTELECT password FFROMROM users WWHEREHERE name LLIKEIKE 'Admin%'
```

2.2.5 - Inyecciones ciegas (Blind SQL Injection)

Las inyecciones ciegas SQL se basan en detectar ciertos cambios que se dan al producirse un error en la consulta: desde redirecciones a otra página hasta evaluar las diferencias de tiempo que se tarda en procesar una consulta correcta de una incorrecta.

Por ejemplo, para hacer un ejemplo sencillo, supongamos que tenemos en base datos los *passwords* de los usuarios guardados en claro y el usuario con derechos de administración su *id es 1* y su *password* es "*SQL*". Si tenemos un campo donde podemos realizar inyecciones ciegas podríamos realizar la siguiente consulta:

```
SELECT * FROM users WHERE id=1 AND password LIKE 'A%';
                            …
SELECT * FROM users WHERE id=1 AND password LIKE 'S%';
```

Las primeras consultas, como no retornaría resultado probablemente el tiempo de respuesta sería elevado (comparado con un retorno positivo). Al llegar al comparar con 'S', como retornaría resultado positivo (recordemos que el *password* es SQL) el tiempo sería reducido.

Al saber que la primera letra es una S ya podríamos probar con la siguiente letra:

```
SELECT * FROM users WHERE id=1 AND password LIKE 'SA%';
                              …
SELECT * FROM users WHERE id=1 AND password LIKE 'SQ%';
```

Tal como pasó antes tenemos que las primeras consultas la respuesta sería lenta comparado con la que nos retorna un resultado positivo (en este caso “SQ”).

2.3 – Prevención

Dadas las graves consecuencias que se pueden dar al sufrir este tipo de ataques se hace de vital importancia aplicar los métodos que sean necesarios para poder evitarlos.

Las acciones básicas a realizar para evitar con un mínimo de garantías este tipo de ataques son dos: el uso de consultas parametrizadas o procedimientos almacenados y filtros de escapado de caracteres.

Las consultas parametrizadas se basan en diferenciar de manera clara las sentencias SQL de los datos de entrada necesarios para llevarlas a cabo.

Usando este tipo de sentencias preparadas nos aseguramos que la finalidad de la consulta será inamovible independientemente de los parámetros de entrada que sean introducidos.

Pongamos un ejemplo de una sentencia preparada en java:

En este caso, si un atacante intentara inyectar en el campo de *username* una sentencia SQL malintencionada, por ejemplo *' OR '1' = '1* la consulta buscaría, literalmente, un usuario el cual su nombre fuera *' OR '1' = '1.*

Los procedimientos almacenados son, en concepto, igual que las consultas parametrizadas: Separar la parte de la consulta SQL de los parámetros necesarios para ejecutarla. La diferencia entre estos dos métodos radica en que los procedimientos almacenados son definidos y guardados en la propia base de datos y luego son llamados por la propia aplicación.

El segundo método a aplicar es el filtrado de los datos introducidos por el usuario. Este método puede dar resultados algo mejores de rendimiento respecto a los otros dos métodos mencionados anteriormente aunque es algo mas inseguro a la hora de proteger nuestra aplicación.

Esta técnica se basa en que cada DBMS acepta uno o varios caracteres de escape específicos para los parámetros de la consulta.

Un atacante aprovecha normalmente que el parámetro que se introduce vía formulario se añadirá a la consulta iniciado con uno de estos caracteres y se concatenará al final otro carácter de escape adicional para acabar de cerrar la consulta. Sabiendo esto añadirá en el campo del formulario mal protegido un carácter adicional para cerrar la consulta que se pretendía ejecutar y añadir otra consulta maliciosa.

La idea de este método es que todo elemento introducido por el usuario debe ser filtrado para poder neutralizar todos estos caracteres de escape para que no actúen como tal dentro de la consulta.

Dado que cada DBMS acepta diferentes tipos de caracteres de escape, existen diferentes funciones de filtrado para cada uno de los diferentes DBMS.

Capítulo 3 – Descripción de la aplicación

A partir de este punto es donde encontraremos información de la aplicación y su desarrollo.

Esta aplicación pretende ser una herramienta de test para desarrolladores, centrada específicamente en los vulnerabilidades de inyección SQL.

La aplicación es simple de diseño pero contiene bastantes opciones. La parte central, que es la que mas destaca, es donde se muestran los resultados. Los mensajes que se muestran como resultado están categorizados por la prioridad del mensaje, tenemos cuatro tipos que, ordenados por importancia, serian: *info*, *warning*, *error* y *critical.* Para poder distinguir cada tipo y dar mas visibilidad se le ha dado un código de color a cada tipo de mensaje.

Encima de los resultados se encuentra el campo que nos permite introducir la URL a examinar y al lado de esta, las opciones extra para el test.

En el campo de la URL hay que destacar que, si se añade una dirección sin ningún tipo de información de envío de formulario, la aplicación intentará deducir los nombres de los campos habituales de formularios y escaneará estos para comprobar si son vulnerables. En caso que en la URL añadamos información de un envío por GET la aplicación directamente testeará estos campos introducidos.

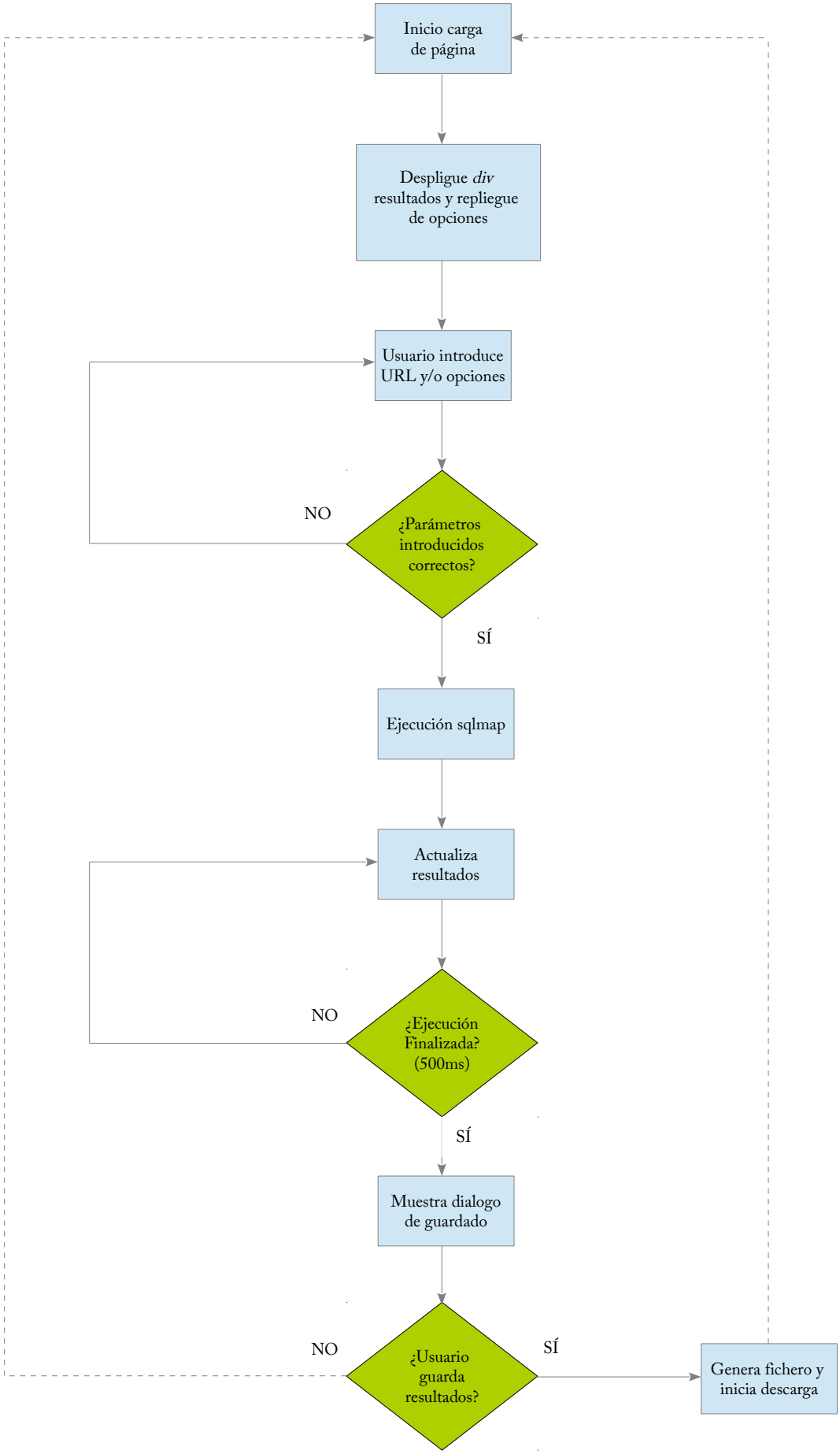
Las opciones extras estarán ocultas por defecto para dejar un diseño mucho mas limpio. Una vez desplegadas podemos seleccionar aquellas que queramos y al ejecutar el test se volverán a ocultar para dejar sitio a los resultados.

Los resultados se van mostrando a medida que se van obteniendo, concretamente se refrescan cada medio segundo.

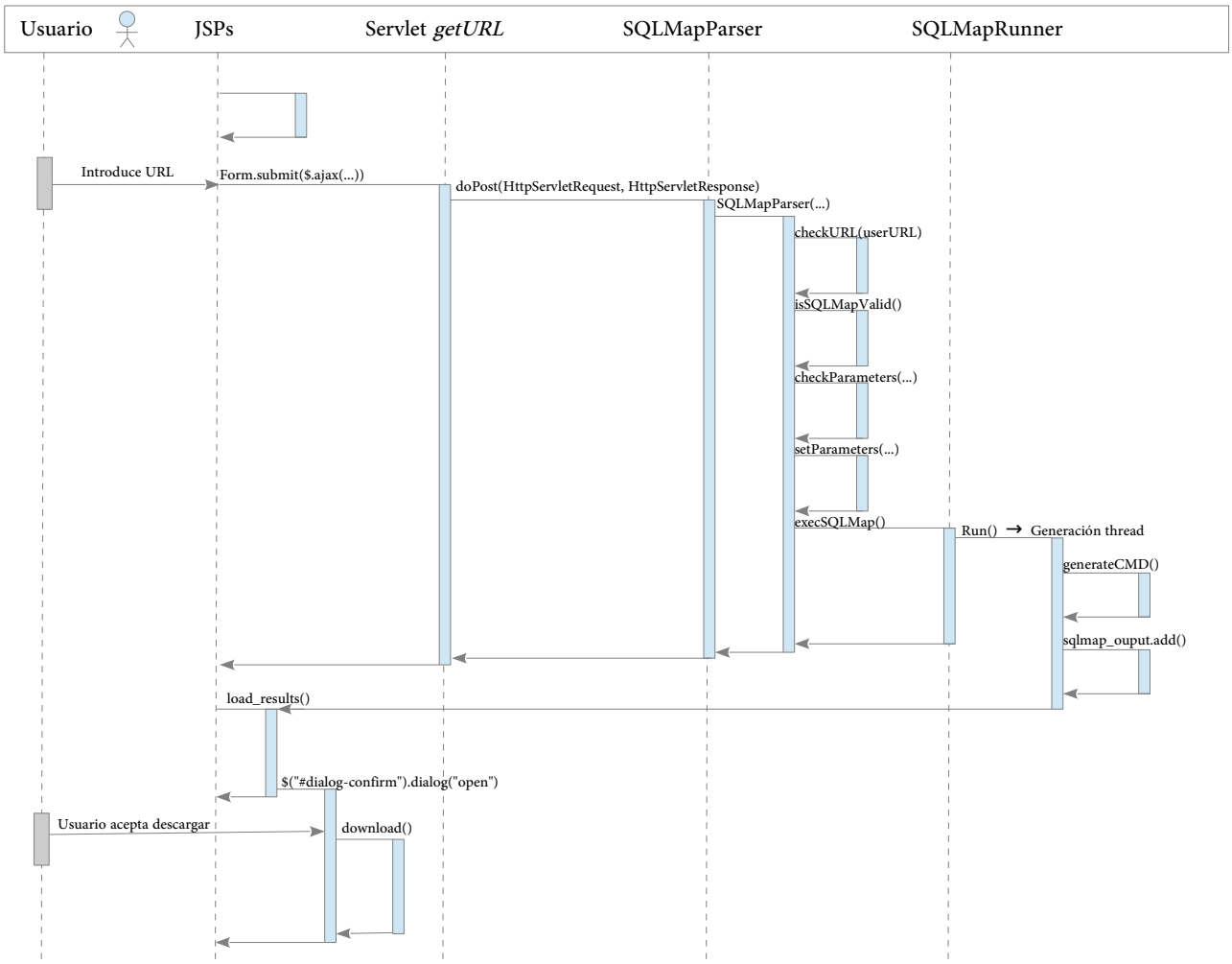
Cuando han finalizado los resultados, se muestra un dialogo para poder escoger si queremos guardarnos los resultados obtenidos en un fichero de texto.

Mostraremos unos diagramas de flujo y secuencia para explicar la aplicación con mas detalle.

3.1 – Diagrama de flujo



3.2 – Diagrama de secuencia



Capítulo 4 – Estructura de la aplicación

Dado el patrón de diseño que se ha escogido para desarrollar esta aplicación, tenemos 3 partes diferenciadas: Modelo, vista y controlador. Pasaremos a explicar como se han realizado cada una de estas tres partes: componentes que la forman y detalles del código dignos de explicar.

4.1– Modelo

La parte modelo está desarrollada en Java y son dos clases diferenciadas: La clase *SQLMapParser* y la clase *SQLMapRunner*.

La primera de las clases se dedica al control de entrada de los datos de usuario, comprobar si el test es realizable y recoger los resultados de la clase *SQLMapRunner*.

La segunda se dedica exclusivamente a llamar al *sqlmap,* gestionar los diferentes parámetros que se ejecutaran y subministrar a la clase *SQLMapParser* con los resultados.

Pasemos a explicar de manera mas detallada cada una de ellas.

4.1.1 - Clase SQLMapParser

Esta clase dispone de 16 variables privadas. Diez de ellas son para almacenar los parámetros que el usuario quiere ejecutar en el test que está realizando. El listado de estos parámetros son *level, risk, data_post, agent, hpp, dbms, dbms_result, optimizations, timeout* y *retries.*

Disponemos de una variable *status* que contiene información del estado del test.

También tenemos dos variables dedicadas a la localización y integridad de la aplicación *sqlmap.* La variable *sqlmap_location* que nos indica donde se encuentra la aplicación *sqlmap* y la variable *sqlmap_hash* que dispone del *hash* de la aplicación en su versión 1.0 que es la que se ha utilizado.

Para hacer la llamada y recibir el resultado de *SQLMapRunner* tenemos dos variables mas: la variable *runner* que es la variable por la cual instanciaremos la clase *SQLMapRunner* y *sqlmap_output* que es donde almacenaremos los resultados.

Finalmente tenemos la variable *url* que es donde se almacenará la URL que el usuario quiere hacer el test.

Constructor

En el constructor haremos la mayoría de comprobaciones. Iniciamos mirando si la URL es válida llamando a la función *checkURL.* Esta busca caracteres sospechosos y, en caso de encontrarlos, la considera insegura. En caso de pasar esta primera comprobación miramos que la aplicación que estamos ejecutando es realmente el *sqlmap* comprobando el *hash* que nos retorna esta con el que sabemos que es correcto. Esta comprobación la realiza la función *isSQLMapValid*. Si pasamos este segundo test miramos la conectividad de la URL subministrada y en caso de ser correcto comprobamos con la función *checkParameters* los diez parámetros que el usuario a seleccionado para ejecutar en el *sqlmap.*

Si hemos conseguido pasar todas las comprobaciones ejecutaremos la función *setParameters* para guardar todos los parámetros introducidos por el usuario, actualizaremos la variable *status* conforme todo está correcto y crearemos una instancia de la clase *SQLMapRunner* para que empieza la ejecución.

En caso de que alguno de los test mencionados anteriormente no fuera valido, la variable *status* se actualizará dependiendo de que test ha dado el error y se cancelará el test informando al usuario.

Funciones

A parte de las mencionadas en el constructor de la clase que se dedican a hacer comprobaciones de la integridad del *sqlmap,* los parámetros introducidos por el usuario y la disponibilidad de la web a testear, tenemos mas funciones dedicadas a otros aspectos:

- función *resetValues* que resetea las diez variables de parámetros del *sqlmap.*
- Funciones *get:* Para todas las variables menos las que definimos el path del *sqlmap* tenemos su respectiva función *get* para retornar el valor. Todas estas funciones son *protected* para que solo puedan acceder a ella tanto la clase propietaria como las clases que forman parte del mismo paquete (en nuestro caso, *SQLMapRunner*).
- Función *URLisReachable* que comprueba la disponibilidad de la web. Dado que muchos servidores no aceptan paquetes *ICMP* no hacemos un *ping,* simplemente hacemos un *GET* de la URL y si recibimos un *http status code* 200 consideramos que hay conectividad.

- Función *getHash.* No es un *get* de la variable *sqlmap_hash.* Esta función hace un calculo del hash de la aplicación que está donde nos indique la variable *sqlmap_location.*

- *Función isSQLMapValid* es la que llama a *getHash* y compara con *sqlmap_hash* para ver si las dos son iguales.

4.1.2 – Clase SQLMapRunner

Esta clase es la encargada de ejecutar el *sqlmap* con los parámetros dados por el usuario en la web y generar los resultados que se irán imprimiendo por pantalla a medida que los vayamos obteniendo.

Esta clase extiende a *Thread* ya que ejecutará el *sqlmap* en un *thread* nuevo para no bloquear la ejecución y poder ir mostrando los resultados a medida que los vamos obteniendo.

Disponemos de cinco variables: las variables con el *path* del *sqlmap* y su *hash* (*sqlmap_location* y *sqlmap*_hash) un *booleano* para indicar la finalización de la ejecución *(isFinished*), una variable con el *output* del *sqlmap* (*sqlmap_output*) y finalmente una variable *parser* que contiene la instancia de *SQLMapParser* que la ha llamado (*parser).*

Constructor

Este constructor solo requiere como argumento una clase *SQLMapParser* que será asignada a la variable *parser.*

Funciones

- Funciones *get:* En esta clase solo tenemos dos funciones *get* de las variables *isFinished* y *sqlmap_output.*

- Función *getHash:* Volvemos a comprobar otra vez que el *sqlmap* que ejecutamos tiene el *hash* que toca, así que esta función calcula el hash de la función que vamos a ejecutar.

- Función *isSQLMapValid:* Aquí es donde comprobamos que la aplicación que ejecutamos es realmente el *sqlmap.*

- Función *generateCMD:* Se encarga de generar, a partir de los parámetros introducidos por el usuario en la web, los equivalentes a los parámetros para ejecutarlos en el *sqlmap.*

- Función *start:* Es un *override* de *thread.* Solo llama a la función *start* de la clase padre.

- Función *run:* Es un *override* de *thread.* Esta función es la encargada de llamar a la función *generateCMD* y ejecutar el *sqlmap.* A medida que se vaya generando *output* lo iremos añadiendo a la variable *sqlmap_output* que es de la cual la vista irá cogiendo los datos para ir mostrándolos por pantalla.

La carpeta WEB-INF/classes es donde están las clases de java *SQLMapParser* y *SQLMapRunner* junto con el código del *servlet* (explicado en el punto 4.2).

4.2 – Controlador

La parte de controlador de la aplicación es un *servlet* de *java* que se la ha dado el nombre de *getURL.* Este controlador es el que recibe la petición del usuario con los parámetros a ejecutar en el *sqlmap* y la URL.

En este *servlet* recibimos los datos enviados por *post* desde la aplicación en el *http servlet request.* A partir de aquí los iremos cogiendo y guardando para crear el objeto *SQLMapParser.* Cuando este sea creado lo guardaremos en una variable de sesión para que luego pueda ser usada para poder mostrar por pantalla los datos.

Dentro de la carpeta WEB-*INF* tenemos el fichero *web.xml* donde definimos tanto el *servlet* como el *mapping* de este.

En la carpeta WEB-INF tenemos la subcarpeta *classes* donde están las clases de java *SQLMapParser* y *SQLMapRunner* y el código del *servlet*.

4.3 – Vista

Esta parte se ha desarrollado con *jsp*, usando *javaserver standard tag library (jstl)* y *expression language (EL),* además de *jquery*, el cual usaremos para cargar diferentes componentes con *ajax.*

La web está estructurada de la siguiente manera: en el directorio raíz tenemos las diferentes páginas con jsp. Concretamente tenemos cuatro: *index.jsp, header.jsp, footer.jsp* y *results.jsp.*

Luego tenemos una carpeta *js* donde están los fichero que contienen el *javascript* que se usa en la web: *jquery-2.0.3.min.js*, *jquery-ui-1.10.3.custom.min.js* y el fichero *main.js* que es el que se ha creado específicamente para controlar la parte cliente de la aplicación.

También tenemos una carpeta *css* donde están los fichero de estilo: tenemos el fichero *css* del estilo principal de la página (*style.css*) y el fichero *jquery-ui-1.10.3.custom.css* junto con la carpeta *images* donde están los ficheros de

imagen necesarios para que *jquery-ui* genere los diálogos.

4.3.1 - Ficheros JSP

- *Header.jsp*

Este fichero es el mas simple. Simplemente tiene el contenido el código de lo que es la cabecera de la web (figura 4-1).

Figura 4-1: Muestra del encabezado

- *Footer.jsp*

En este fichero tenemos el pie de página. En este tenemos una mención a que la web usa *sqlmap,* y se da crédito a los autores de este y un enlace a la web del programa (figura 4-2).

Figura 4-2: Muestra del pie de página

- *Results.jsp*

Aquí es donde se irán imprimiendo los resultados generados por el *sqlmap*.

Usando *JSTL* y *EL* imprimimos el contenido de la variable *sqlmap_output* en ese momento. Para dar un poco de estilo a la salid del *sqlmap* aquí es donde añadiremos un *tag* de *span* donde añadiremos una clase dependiendo del tipo de mensaje que nos retorne el *sqlmap: critical, error, warning* o *info* (figura 4-3).

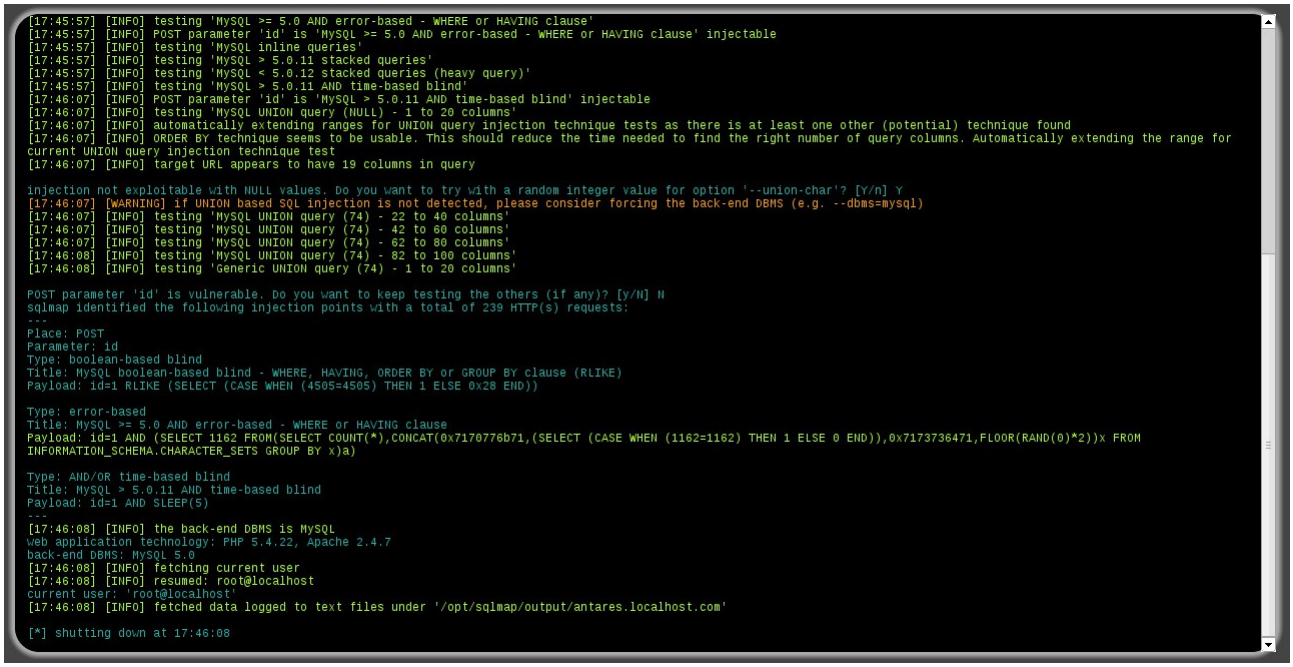


Figura 4-3: ejemplo de resultados

- Index.jsp

Esta es la página principal donde se muestra toda la información.

La cabecera y el pie de página se añaden haciendo *includes* en sus respectivos *divs:* para header.jsp lo insertaremos en el *div* "*header*" y para *footer.jsp* en el *div footer.*

Debajo de la cabecera tendremos un formulario formado por un *input url (html 5.0)* que añade otro nivel de control de los datos introducidos por el usuario ya que automáticamente limita que la entrada en estos tipos de campos en el formulario sean *urls*. Al lado del campo *url* tendremos un botón que, al clicar, se despliega mediante *ajax* mas opciones del formulario (figura 4-4). Las opciones disponibles son: *data post, level, risk, random agent, http parameter pollution, timeout, retries, dbms dbms retrieve options* y *optimizations.*

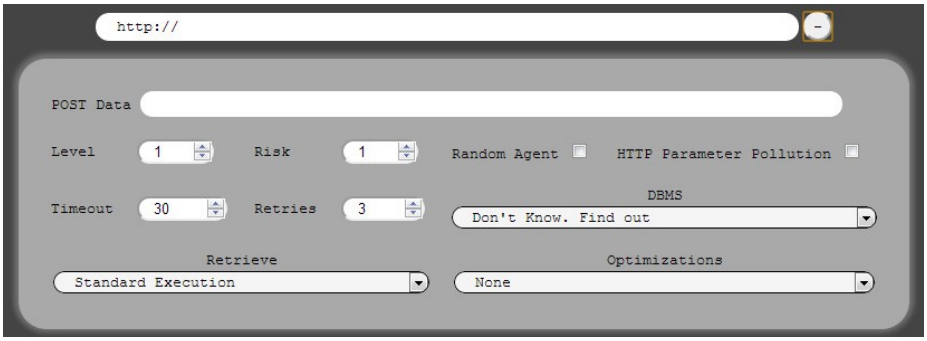


Figura 4-4: Campos url y opciones del formulario

Debajo de los parámetros de ejecución del *sqlmap* disponemos de la zona donde se muestran los resultados que genera *results.jsp.* Estos se cargan mediante *ajax* en el *div results,* que originalmente se encuentra vacío. Esta carga se hace cada medio segundo para poder mostrar los datos que genera *sqlmap* de manera inmediata. Esta parte se explica en mas detalle en este mismo punto en la explicación del fichero *main* de javascript.

Cuando se finaliza la ejecución de *sqlmap* mostramos un dialogo para que el usuario pueda elegir si quiere guardar el resultado obtenido (figura 4-5). En caso afirmativo generaremos mediante *javascript*, en el propio cliente, los resultados generados. Usamos *jquery-ui* [9] para generar este dialogo.

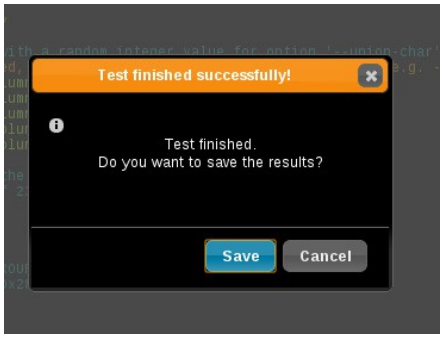


Figura 4-5: Dialogo de descarga de resultados

4.3.2 - Ficheros Javascript

De los tres ficheros de javascript que tiene la aplicación, explicaremos *main.js,* que es el creado a propósito para el desarrollo de esta aplicación. Omitiremos la explicación de *jquery* y *jquery-ui* debido al tamaño y complejidad de estos archivos y son suficientemente usados y conocidos como para encontrar gran cantidad de información sobre estos.

Main.js

En este fichero tenemos las diferentes funciones que se ejecutan en la web. Pasemos a explicarlas.

- Función de carga de la página: dentro de *window.load* tenemos una función que se ejecutará al cargarse la página. Dentro de esta función realizamos la preparación de la mayoría de elementos: ocultamos el desplegable con las opciones adicionales para el test y desplegamos el *div* donde se encuentran los resultados, inicializamos el dialogo que se mostrará al final la ejecución del test, y preparamos el formulario. Pasemos a explicar esta preparación de manera mas profunda:

  Mediante *jquery ajax* preparamos el formulario para definir que los datos se enviaran a nuestro *servlet getURL, y* que los parámetros a enviar los preparará la función *getData* (explicada mas adelante) y que al finalizar el envió de este formulario (*success*) realizaremos un intervalo cada medio segundo, y que en

cada uno de estos intervalos se ejecute la función *load_results,* que explicamos mas adelante, pero básicamente se encarga de cargar los resultados y controlar que la ejecución de *sqlmap* ha finalizado para detener el intervalo.

- Función *moreOptions*: Simplemente se encarga de hacer el despliegue de las opciones extra del test.

- Función *load_results:* En esta función es donde cargamos, en el *div results,* el contenido de *results.jsp.* Si no tuviéramos un control de finalización de los resultados, la página estaría recargando el *div* cada medio segundo de manera infinita. Lo que hacemos es mirar el contenido de la web y cuando vemos que *sqlmap* muestra un mensaje de *shutting down* paramos la recarga porqué sabemos que ha finalizado.

  Al detectar el fin de la ejecución de *sqlmap,* mostramos el dialogo para dar la opción al usuario de guardar los resultados del test.

- Función *download:* Esta función es la que se encarga de generar el fichero de texto de los resultados obtenidos. Dado que no se generan de la misma manera dependiendo del sistema operativo donde se ejecute el navegador se distingue entre *windows* y *linux*. La idea, aún así, es la misma independientemente del S.O.: coger el contenido del *div results* y añadirlas en un fichero de texto. La ventaja de hacerlo de este modo en vez de hacerlo en la parte servidor es que es el propio cliente el que genera el fichero y el servidor se ve descargado de carga de proceso y ahorro de espacio en disco al no tener que almacenar los resultados de todas las peticiones.

- Función *encodeString:* Usada en la función *download.* Dependiendo del S.O. realizaremos una acción o otra para conseguir generar el mismo fichero de texto. Lo que se hace es eliminar la mayoría de saltos de linea y espacios de mas que genera el *sqlmap* para no hacer un fichero excesivamente grande. En *windows* simplemente podemos quitar saltos de linea sobrantes y eliminar *tags* ya que tenemos que usar directamente el código *html,* mientras que en *linux/MAC OS* podemos coger únicamente el texto, y eliminar saltos de linea y espacios sobrantes.

- Función *slideResults*: función meramente estética. Despliega el *div* de resultados.

- Función *getData*: En esta función es donde cogemos todos los datos que enviaremos al *servlet* para generar el test con *sqlmap.* Crearemos el listado de parámetros en formato *JSON,* serializamos la url y añadimos el resultado de ejecutar la función *param* de *jquery* a la lista en formato *JSON* recién generada.

Capítulo 5 – Implantación

5.1 - Proceso Instalación del Entorno de Trabajo

Crearemos una máquina virtual usando VMPlayer versión 5.0.2 [10]. En la máquina virtual que se ha creado se instaló un Ubuntu Server 12.04 LTS de 64 bits.

5.1.1 - Instalación JDK

- Versión: Oracle Java 64-bit 1.7.0_40-b43 [11]

Descomprimimos el tar.gz en la carpeta /opt/jdk1.7.0_40. Creamos un enlace simbólico débil a esta carpeta en /opt/jdk

Creamos un enlace simbólico débil en /usr/bin/java que apunte a /opt/jdk/bin/java.

Creamos un segundo enlace simbólico débil en /usr/bin/javac que apunte a /opt/jdk/bin/java.

5.1.2 - Instalación Tomcat

- Versión: Apache Tomcat 7.0.42 [12]

Descomprimimos el zip en /opt/apache-tomcat-7.0.42 y creamos un enlace simbólico débil que apunte a esta carpeta en /opt/tomcat.

Securización del *tomcat*

- Eliminación de aplicaciones innecesarias

Eliminaremos toda aplicación no necesaria del tomcat así que borraremos, dentro de la carpeta [1] /opt/tomcat/webapps, las carpetas: *docs, example, host-manager, manager* y *ROOT.*

- Ejecución con usuario estándar

Se ha creado un usuario especifico únicamente para ejecutar el *tomcat* que tenga privilegios básicos. Así evitaremos ejecutarlo como *root.*

Para realizar esta operación se ha instalado la aplicación JSVC, de esta manera podemos realizar dos acciones: iniciar el proceso del tomcat como un usuario estándar y configurar *tomcat* como un servicio (*daemon*).

Para instalar el *jsvc* (*commons-daemon*), el propio *tomcat* proporciona el código fuente para que cada usuario se lo pueda compilar. Iremos a la carpeta /opt/tomcat/bin y descomprimiremos el fichero *commons-daemon-native.tar.gz*. En la carpeta generada haremos un configure especificando la localización del *jdk*. Una vez realizado haremos un *make* y el fichero *jsvc* generado lo moveremos a la carpeta /opt/tomcat/bin.

Realizado esto haremos un cambio de propietario recursivo a toda la carpeta *tomcat* poniendo como nuevo propietario nuestro usuario y grupo.

Para acabar la configuración crearemos el *script* que arranque el *tomcat* como *daemon* ejecutado por el usuario *tomcat*.

El fichero será /etc/init.d/tomcat y contendrá lo siguiente:

```
#!/bin/sh
#
# tomcat           Utilizing the jsvc start and stop the servlet engine.
#
# chkconfig: 345 99 10
# description: Starts and stops the tomcat servlet engine.
# Source function library.
JAVA_HOME=/opt/jdk/
JAVA_OPTS='-Xms256m -Xmx1024m -XX:MaxPermSize=512m -server -Djava.awt.headless=true'
CATALINA_HOME=/opt/tomcat
DAEMON_HOME=/opt/tomcat/bin/
TOMCAT_USER=tomcat
TOMCAT_GROUP=tomcat
TOMCAT_PID=/var/run/jsvc_alfresco.pid
TMP_DIR=/opt/tomcat/temp
CATALINA_OPTS=
CLASSPATH=$JAVA_HOME/lib/tools.jar:$CATALINA_HOME/bin/commons-daemon.jar:
$CATALINA_HOME/bin/bootstrap.jar:$CATALINA_HOME/lib/tomcat-juli.jar
CATALINA_WORK_DIR=$CATALINA_HOME/work
# To get a verbose JVM
```

```
#-verbose \
# To get a debug of jsvc.
#-debug \
WAIT_MAX=2
case "$1" in
start)
########################################
# Start Tomcat
########################################
echo -n "Starting Tomcat: "

$DAEMON_HOME/jsvc -user $TOMCAT_USER -pidfile $TOMCAT_PID -home $JAVA_HOME -jvm server
$JAVA_OPTS -Dcatalina.home=$CATALINA_HOME -Djava.io.tmpdir=$TMP_DIR -outfile
$CATALINA_HOME/logs/catalina.out -errfile '&1' $CATALINA_OPTS -cp $CLASSPATH
org.apache.catalina.startup.Bootstrap
sleep 2s
if [ -f $TOMCAT_PID ]; then
            echo "success"
else
            echo "failure"
fi
echo
;;
stop)
########################################
# Stop Tomcat
########################################
echo -n "Stopping Tomcat: "
PID=`cat $TOMCAT_PID`
kill $PID 2>/dev/null 1>/dev/null
if [ `ps -p ${PID} > /dev/null 2>&1` ]; then
            echo "failure"
else
            echo "success"
            rm -f $TOMCAT_PID > /dev/null 2>&1
fi
echo
;;
flush)
########################################
# flush Tomcat
########################################
   echo -n "Stopping Tomcat: "
   PID=`cat $TOMCAT_PID`
   kill $PID 2>/dev/null 1>/dev/null
   if [ `ps -p ${PID} > /dev/null 2>&1` ]; then
               echo "failure"
   else
               echo "success"
               rm -f $TOMCAT_PID > /dev/null 2>&1
   fi
   echo
```

```
   echo -n "Flushing Tomcat CACHE ($CATALINA_WORK_DIR) : "
   rm -rf $CATALINA_WORK_DIR
   if [ -e $CATALINA_WORK_DIR ]; then
                echo "failure"
                echo
                exit 1;
   fi
   mkdir $CATALINA_WORK_DIR
   chown $TOMCAT_USER.$TOMCAT_GROUP $CATALINA_WORK_DIR
   if [ ! -e $CATALINA_WORK_DIR ]; then
                echo "failure"
                echo
                exit 1;
   fi
   echo "success"
   echo
   ;;
*)
########################################
# Altres
########################################
   echo "Usage: $0 {start|stop|flush}"
   exit 1;;
esac
exit 0
```

5.2 – Despliegue de la aplicación

Tenemos generado un fichero *war* de despliegue que será el que usaremos para hacer la instalación de la aplicación. Pero primero empezamos por preparar otros pasos previos.

Instalación *SQLMAP*

No requiere ningún tipo de instalación, simplemente bajarnos la versión 1.0 que será la que tenga el *hash* que nuestra aplicación tiene implementado. Otras versiones no coincidirán en *hash* y no funcionaran. Simplemente hemos de tener cuidado de descomprimir y copiar el contenido en una carpeta en **/opt/sqlmap**

Hay que tener en cuenta que la carpeta entera ha de tener los permisos necesarios para ser ejecutado por el usuario que ejecute el *tomcat.* Así que procederemos a ejecutar el comando:

chown -R tomcat:tomcat /opt/sqlmap

Instalación del fichero war

Simplemente movemos el fichero *javaSQLMap-tomcat.war* en la carpeta /opt/tomcat/webapps. Si miramos los *logs* del tomcat (*/opt/tomcat/logs/catalina.out*) deberíamos ver los mensajes conforme se ha desplegado la aplicación.

Si por lo que sea vemos que el desplegado no ha funcionado hay que comprobar que el *war* tiene, como mínimo, permisos de lectura para el usuario *tomcat.* Si aun así no funcionase se podría hacer un despliegue manual descomprimiendo el *war* en la carpeta */opt/tomcat/webapps/javaSQLMap-tomcat*

Llegados a este punto la aplicación se encuentra desplegada. Podemos acceder a ella, des de un navegador, por la url:

http://localhost:8080/javaSQLMap-tomcat

5.3 – Resultados

Se ha conseguido realizar con éxito una aplicación que realiza tests de inyecciones SQL a una URL dada pudiendo definir cierto numero de parámetros suficientemente interesantes y útiles a la hora de realizar pruebas sobre este tipo de vulnerabilidad.

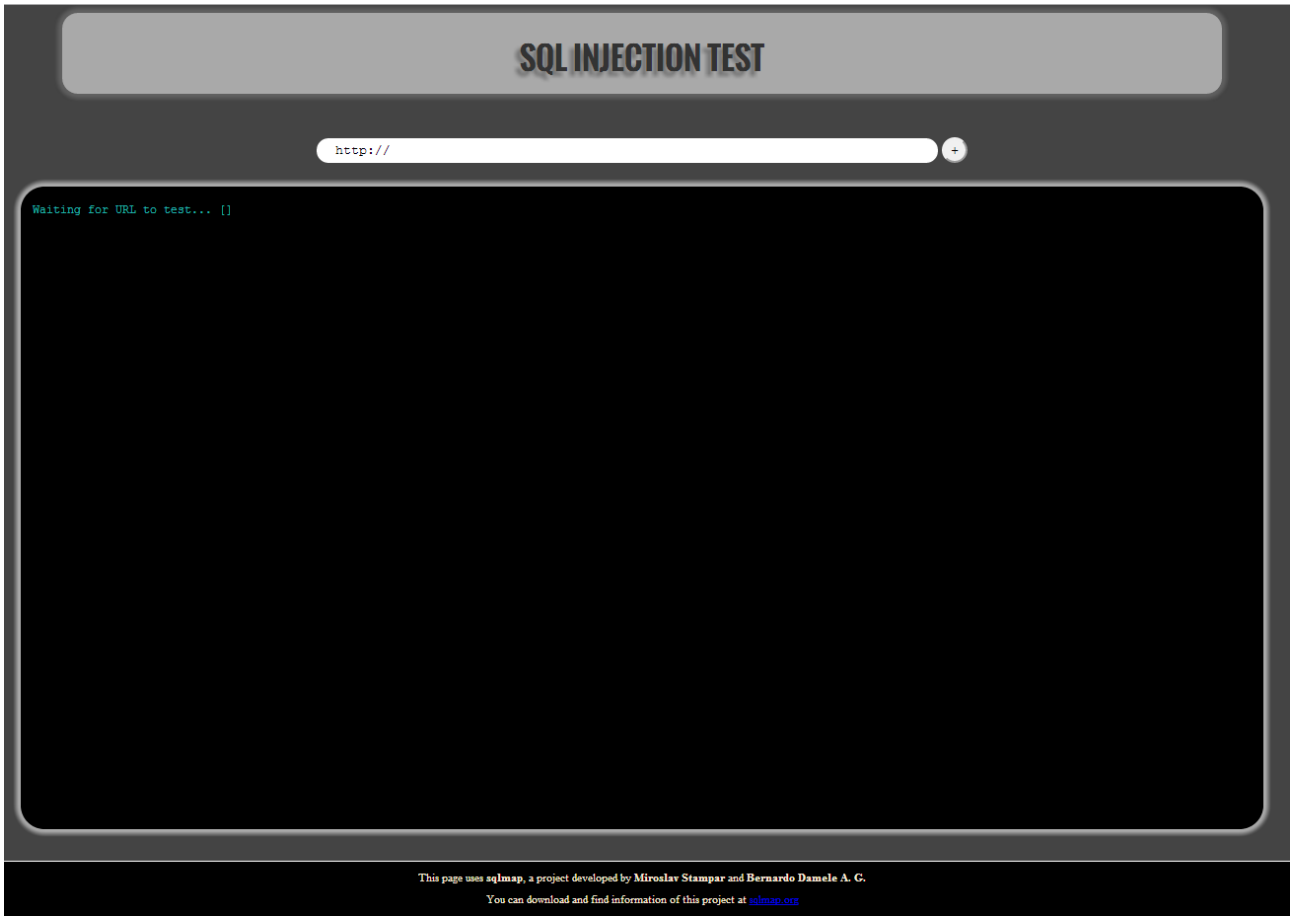
Se ha conseguido también otro de los objetivos planteados que era conseguir mostrar los resultados en tiempo real desde la web tal como lo veríamos si lo estuviéramos ejecutando a mano desde un terminal.

Finalmente otro de los objetivos conseguidos era ofrecer al usuario la posibilidad de guardarse el resultado obtenido en el test en formato texto para que este pueda disponer de esta información.

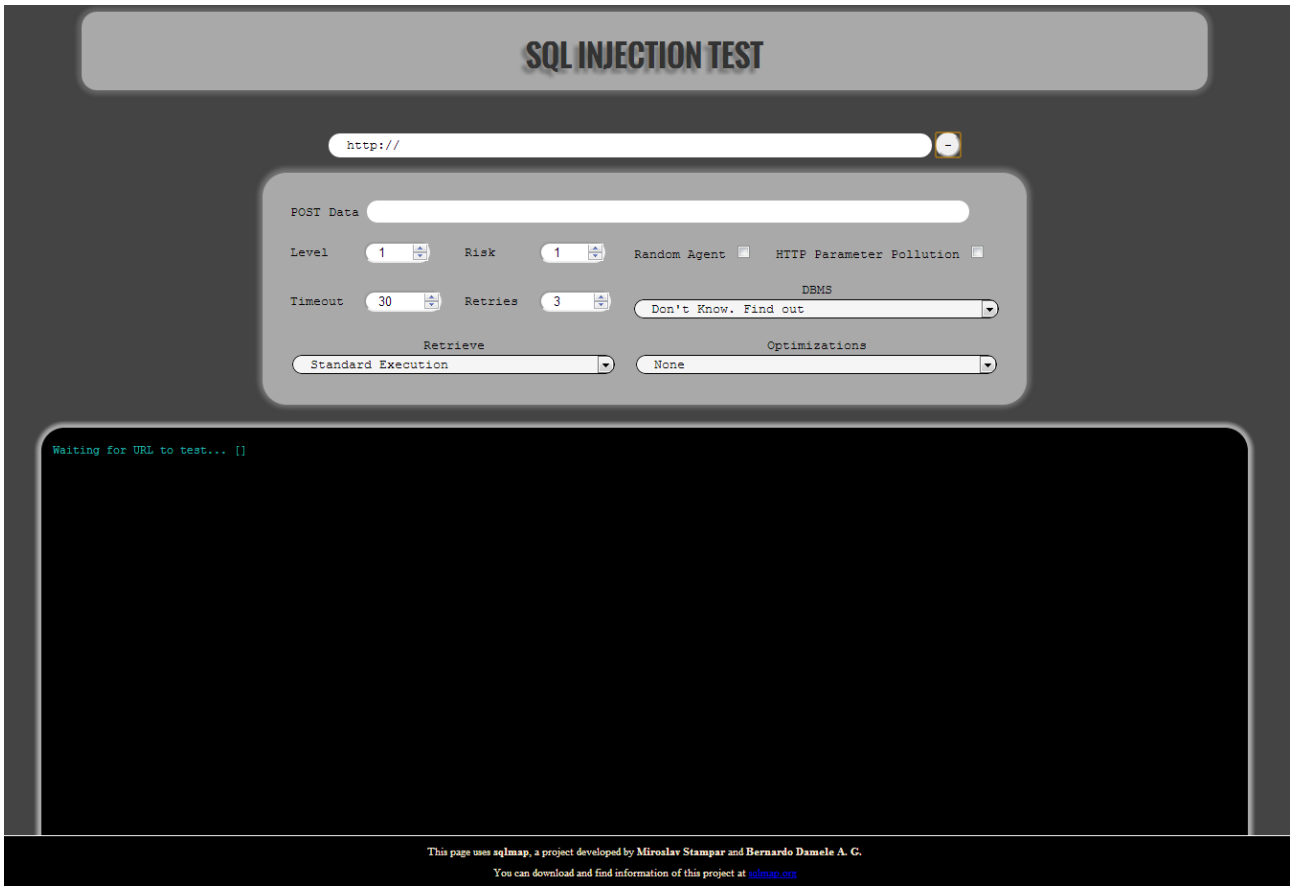
De manera personal he de comentar que no soy una persona demasiado hábil a la hora de dar estilo a las páginas web, pero considero que el estilo conseguido es simple pero atractivo. Quizás el exceso de colores recarga un poco la vista, pero la idea era conseguir poder distinguir de un vistazo que partes de los resultados son mas importantes y creo que se ha conseguido.

Pasemos a mostrar unas capturas de pantalla de los resultados obtenidos:

1. Página recién cargada – Opciones avanzadas ocultas

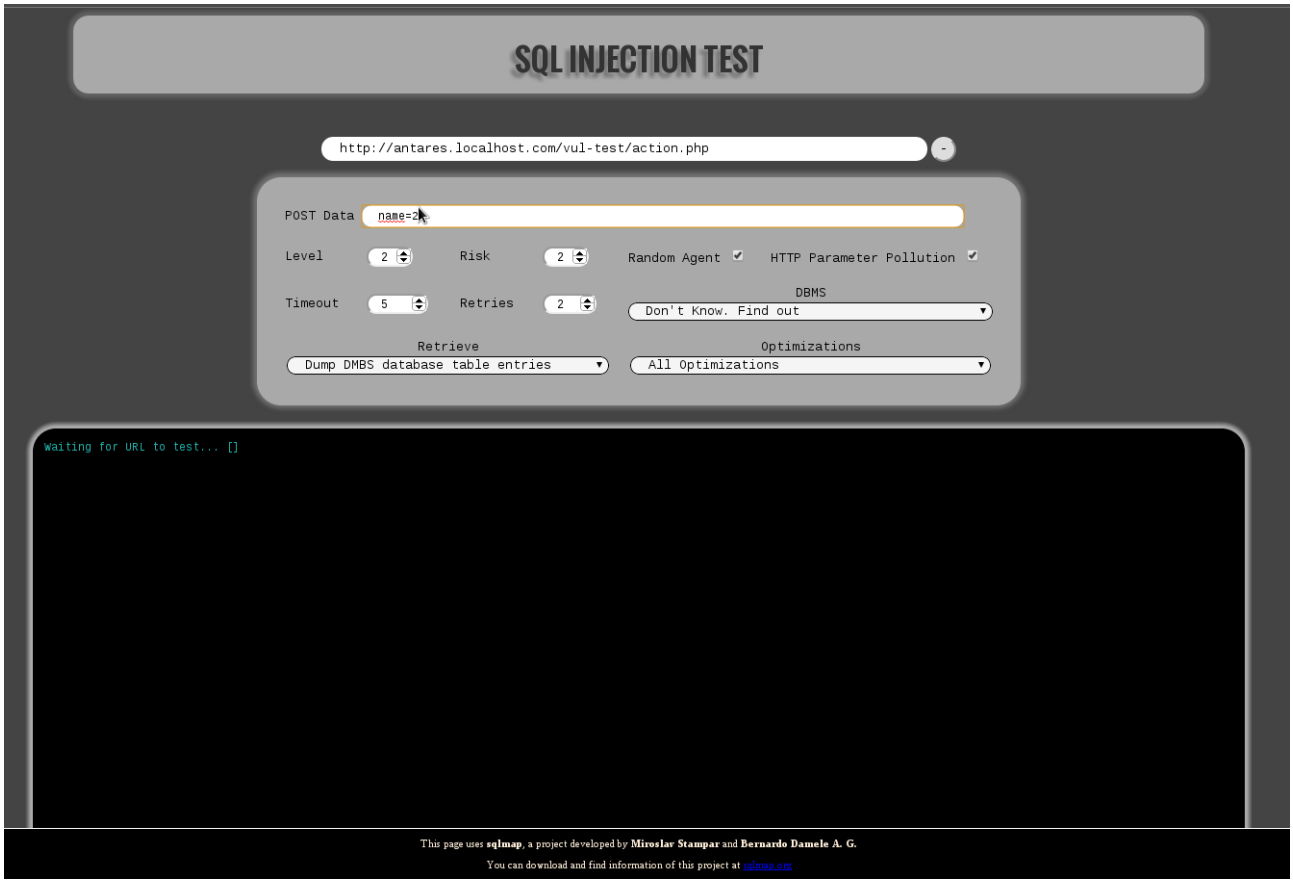


2. Página recién cargada – Opciones avanzadas desplegadas valores por defecto



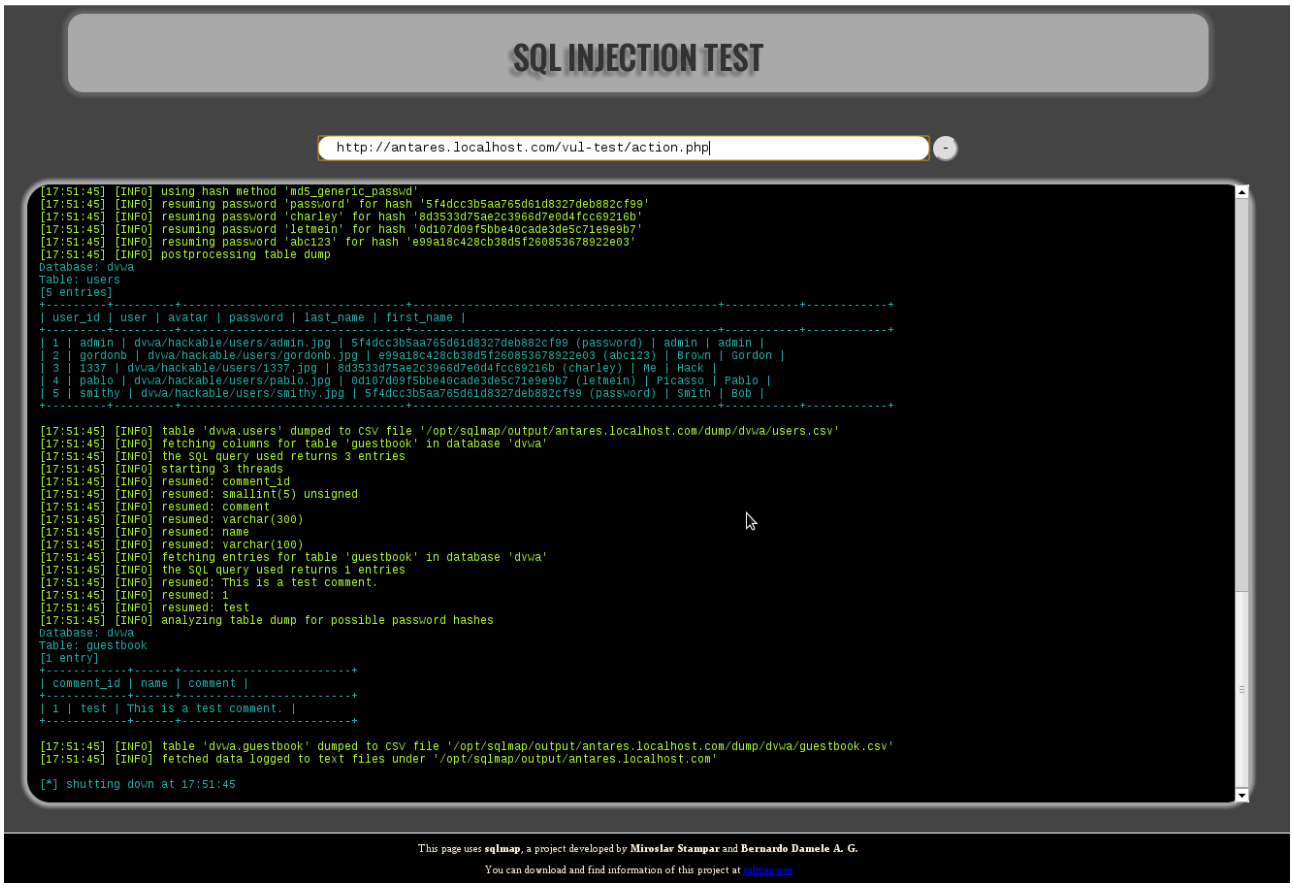
3. Ejemplo de modificación de opciones avanzadas

Podemos ver como hacemos un test donde definimos la variable a escanear por *post*, modificando valores de *timeout* y *retries,* especificando que queremos de retorno un *dump* del contenido de las tablas de la base de datos con un *level* 2 (de 5) y un riesgo 2 (de 3) haciendo las peticiones con un agente aleatorio y intentando buscar si se puede hacer *http parameter pollution.*

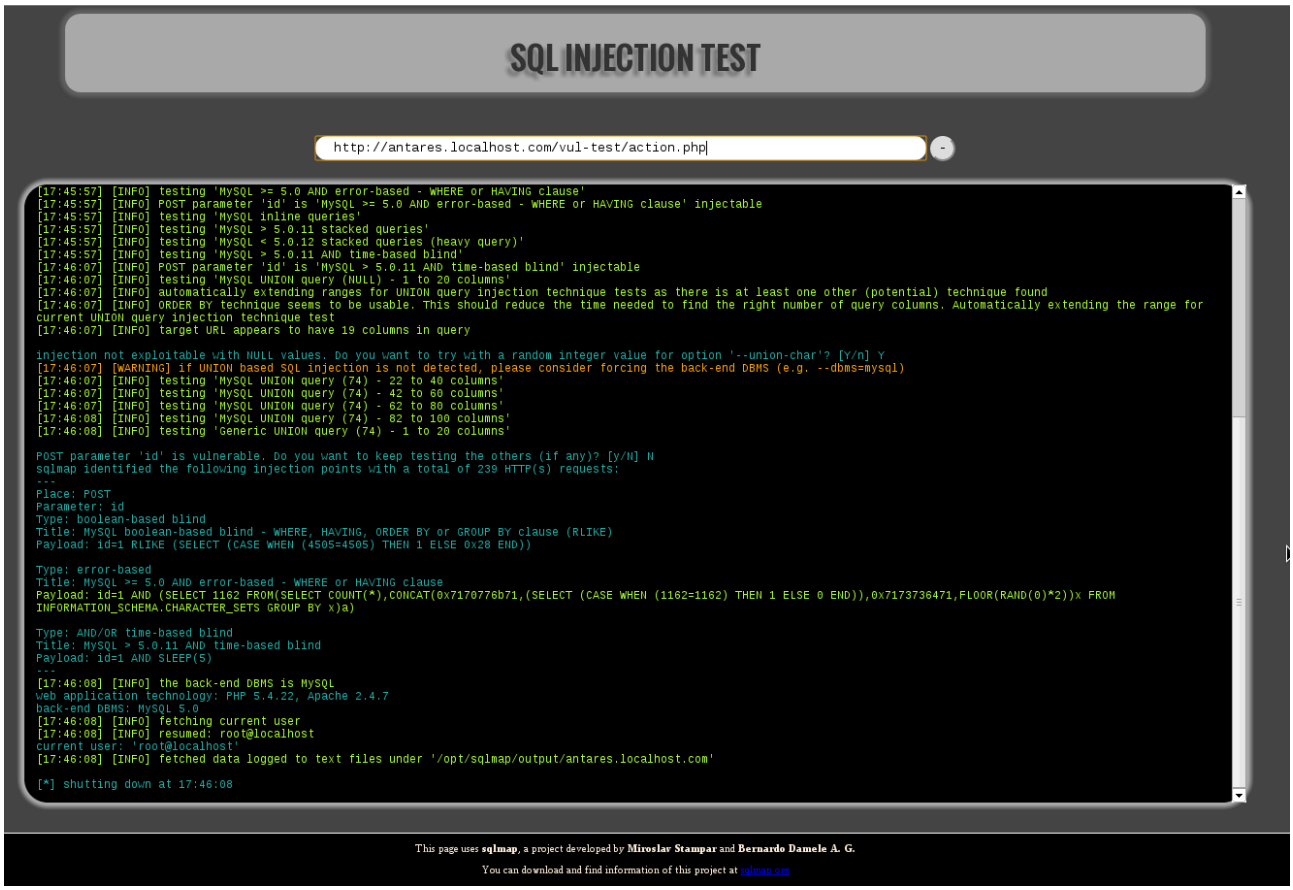


4. Resultado del test anterior

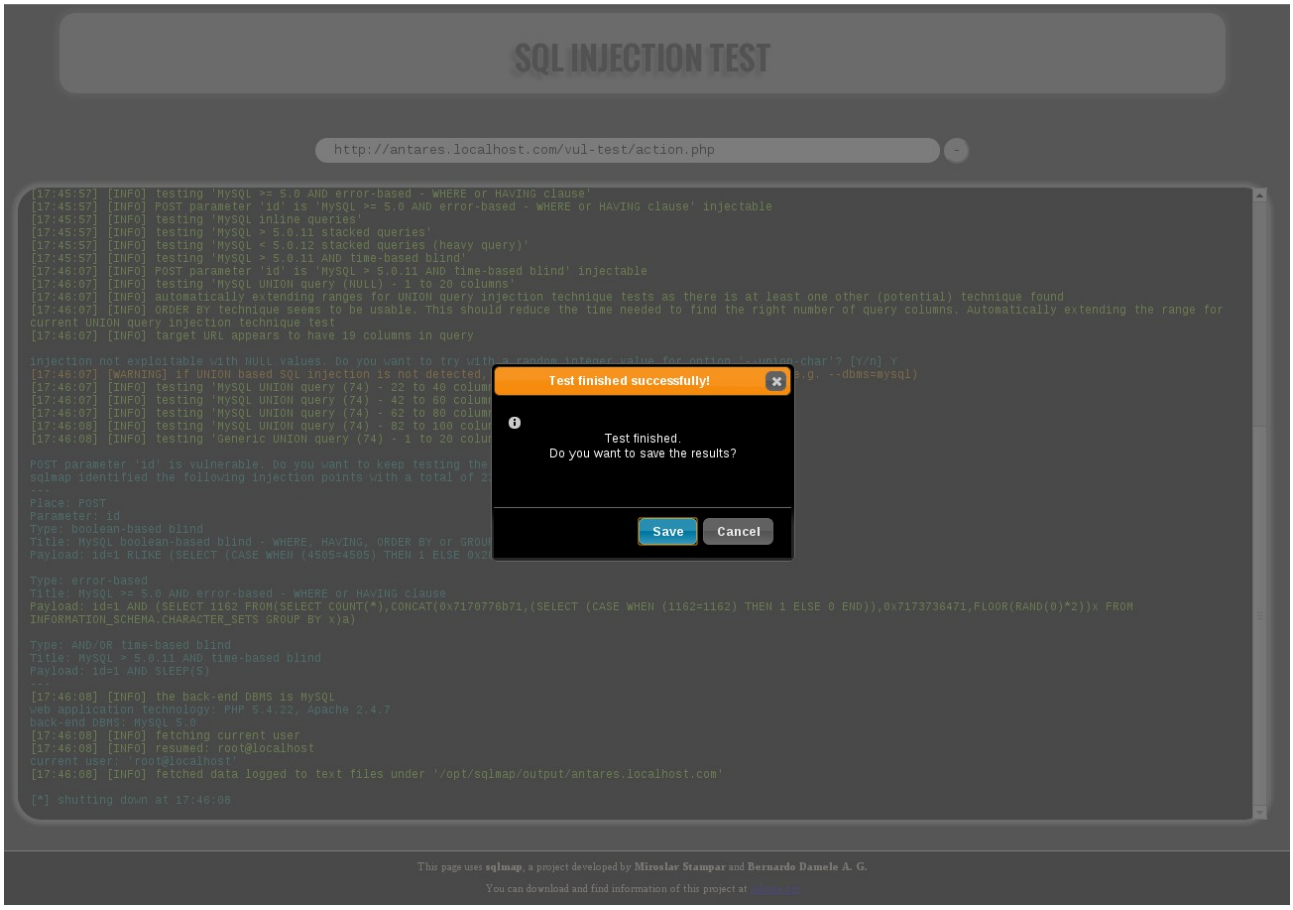
Podemos apreciar como se ven las entradas de las tablas



5. Ejemplo de un test donde buscamos el usuario actual de la base de datos



6. Ejemplo del dialogo de descarga de resultados



7. Captura de pantalla de un fichero de resultados

results_http---anta...05T16-54-35.780Z.txt

```
do you want to crack them via a dictionary-based attack? [Y/n/q] Y
[17:54:33] [INFO] using hash method 'md5_generic_passwd'
[17:54:33] [INFO] resuming password 'password' for hash '5f4dcc3b5aa765d61d8327deb882cf99'
[17:54:33] [INFO] resuming password 'charley' for hash '8d3533d75ae2c3966d7e0d4fcc69216b'
[17:54:33] [INFO] resuming password 'letmein' for hash '0d107d09f5bbe40cade3de5c71e9e9b7'
[17:54:33] [INFO] resuming password 'abc123' for hash 'e99a18c428cb38d5f260853678922e03'
[17:54:33] [INFO] postprocessing table dump
Database: dvwa
Table: users
[5 entries]
+---------+---------+-------------------------------+-----------------------------------------+-----------+------------+
| user_id | user| avatar| password| last_name | first_name |
+---------+---------+-------------------------------+-----------------------------------------+-----------+------------+
| 1| admin| dvwa/hackable/users/admin.jpg| 5f4dcc3b5aa765d61d8327deb882cf99 (password) | admin| admin|
| 2| gordonb | dvwa/hackable/users/gordonb.jpg | e99a18c428cb38d5f260853678922e03 (abc123)| Brown| Gordon|
| 3| 1337| dvwa/hackable/users/1337.jpg| 8d3533d75ae2c3966d7e0d4fcc69216b (charley)| Me| Hack|
| 4| pablo| dvwa/hackable/users/pablo.jpg| 0d107d09f5bbe40cade3de5c71e9e9b7 (letmein)| Picasso| Pablo|
| 5| smithy| dvwa/hackable/users/smithy.jpg| 5f4dcc3b5aa765d61d8327deb882cf99 (password) | Smith| Bob|
+---------+---------+-------------------------------+-----------------------------------------+-----------+------------+
[17:54:33] [INFO] table 'dvwa.users' dumped to CSV file '/opt/sqlmap/output/antares.localhost.com/dump/dvwa/users.csv'
[17:54:33] [INFO] fetching columns for table 'guestbook' in database 'dvwa'
[17:54:33] [INFO] the SQL query used returns 3 entries
[17:54:33] [INFO] starting 3 threads
[17:54:33] [INFO] resumed: comment_id
[17:54:33] [INFO] resumed: smallint(5) unsigned
[17:54:33] [INFO] resumed: comment
[17:54:33] [INFO] resumed: varchar(300)
[17:54:33] [INFO] resumed: name
[17:54:33] [INFO] resumed: varchar(100)
[17:54:33] [INFO] fetching entries for table 'guestbook' in database 'dvwa'
[17:54:33] [INFO] the SQL query used returns 1 entries
[17:54:33] [INFO] resumed: This is a test comment.
[17:54:33] [INFO] resumed: 1
[17:54:33] [INFO] resumed: test
[17:54:33] [INFO] analyzing table dump for possible password hashes
Database: dvwa
Table: guestbook
[1 entry]
+------------+------+-------------------------+
| comment_id | name | comment|
+------------+------+-------------------------+
| 1| test | This is a test comment. |
+------------+------+-------------------------+
[17:54:33] [INFO] table 'dvwa.guestbook' dumped to CSV file '/opt/sqlmap/output/antares.localhost.com/dump/dvwa/guestbook.csv'
[17:54:33] [INFO] fetched data logged to text files under '/opt/sqlmap/output/antares.localhost.com'
[*] shutting down at 17:54:33
```

Texto plano

Capítulo 6 – Conclusiones

6.1 – Conclusiones

Respecto a los objetivos fijados podemos decir que se han cumplido, aunque también se debe remarcar que se ha notado una falta de tiempo considerable como para poder cumplir la planificación prevista del proyecto. De hecho, al haberse alargado el tiempo previsto para conseguir realizar el mostrado de los resultados en tiempo real, el resto de tareas se han ido retrasando.

Los resultados obtenidos son bastante satisfactorios ya que las opciones de ejecución del test que se han conseguido añadir son bastante útiles todo y que, tal como se comenta en el punto siguiente, habría sido interesante haber podido añadir mas opciones ya que la herramienta usada de fondo (*sqlmap)* lo permitía.

Después de acabar el proyecto y haber visto alguna otra aplicación dedicada a la tarea que aquí se ha realizado, le daría una segunda oportunidad a la aplicación de *zed attack proxy.* No porqué sea mas potente que *sqlmap,* simplemente porque dispone de una *api* de *java* que nos permitiría ejecutarla sin necesidad de usar comandos del sistema como tenemos que hacer ahora con *sqlmap.*

Siendo realistas, esta es una aplicación que no podría ser colgada al público tal como esta ahora y con la arquitectura que se pensó, ya que la ejecución simultanea de varios tests con niveles altos y riesgo elevado implican un uso bastante elevado de CPU que llevaría a que el servidor probablemente se saturara. Teóricamente se podría montar un pequeño clúster de *sqlmaps* pero se tendría que modificar la aplicación para que en vez de buscar en local la aplicación, fuera a un enlace con balanceo de carga para redirigir a una o otra máquina del clúster. Ni siquiera se ha planteado porqué personalmente dudo que una aplicación de este estilo tuviera tal demanda como para plantearnos este esquema.

A modo de conclusión final me gustaría volver a remarcar que el objetivo fijado se ha cumplido, ya que se ha conseguido desarrollar una aplicación web que permite realizar test bastante complejos de manera relativamente sencilla, obteniendo resultados tan completos como el propio usuario quiera obtener.

6.2 – Trabajo futuro y puntos de mejora

Como nota personal me gustaría añadir que se me ha quedado corto el tiempo para desarrollar todo lo que me habría gustado hacer. *Sqlmap* ofrece muchísimas opciones y parámetros muy interesantes que se me han quedado en el tintero por falta de tiempo y quizás, ya de forma personal y como *hobby,* las acabaré añadiendo en un futuro.

Puntos interesantes para trabajar en ellos en futuro son los siguientes:

- Seria posible añadir un *tor proxy* [13] y hacer que *sqlmap* use este para enviar las peticiones, anonimizando por completo el test. No tengo muy claro que en una aplicación con un número considerable de usuarios concurrentes pudiera ofrecer un rendimiento decente con tantas peticiones contra *tor* a la vez. Debería estudiarse primero.

- Otra opción que no se ha podido implementar por falta de tiempo es las opciones para poder especificar autenticación *http*. La mayoría de páginas que están disponibles para hacer pruebas como *webgoat* [14] o *damn vulnerable web application* [15] requieren de autenticación por web, así que, para poder hacer las diferentes pruebas, muchas veces me he visto forzado a hacer una pequeña web que haga de *bypass* de la autenticación.

- Una posible mejora seria crear un pequeño fichero de configuración que permitiera definir de manera sencilla (y no dentro del código como se tendría que hacer ahora) el *path* del directorio del *sqlmap* y el *hash* de este por si se actualiza la versión. Obviamente para hacerlo seguro se tendría que mirar de ponerlo con los permisos y *owners* adecuados para poner las cosas difíciles ante un ataque.

Links

[1] https://www.owasp.org/

[2] https://www.owasp.org/index.php/Top_10_2013-Top_10

[3] http://sqlmap.org

[4] http://wapiti.sourceforge.net/

[5] https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

[6] http://code.google.com/p/fatcat-sql-injector/

[7] http://www.oracle.com/technetwork/java/javaee/jsp/index.html

[8] http://jquery.com/

[9] http://jqueryui.com/

[10]https://my.vmware.com/web/vmware/free#desktop_end_user_computing/vmware_player/5_0|PLAYER-502|product_downloads

[11]http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html?ssSourceSiteId=otnes

[12] http://tomcat.apache.org/download-70.cgi

[13] https://www.torproject.org/

[14] https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

[15] http://www.dvwa.co.uk/