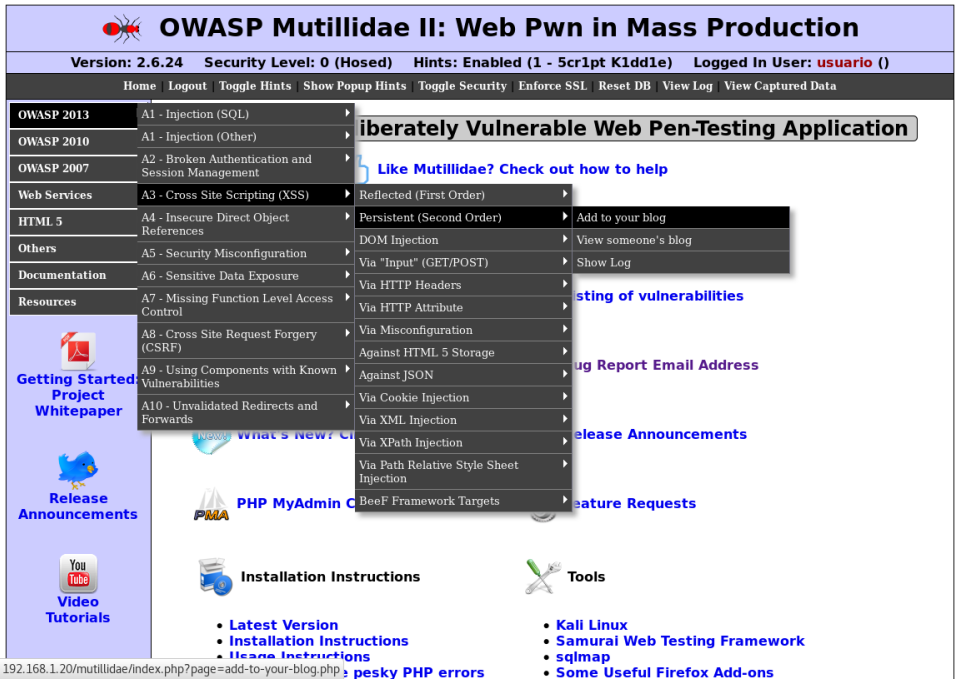
ATAQUE 1

A) Nombre del ataque

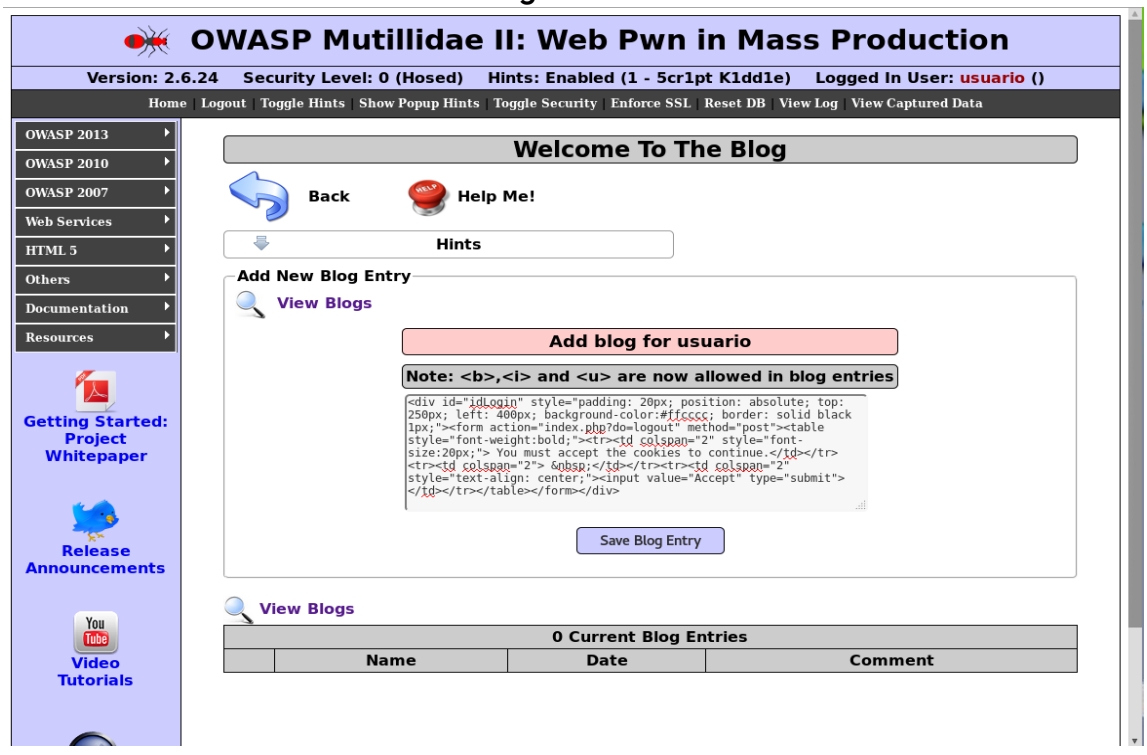
A3 - Cross-Site Scripting (XSS) Persistente (de segundo orden) - Inyección HTML

B) Pasos

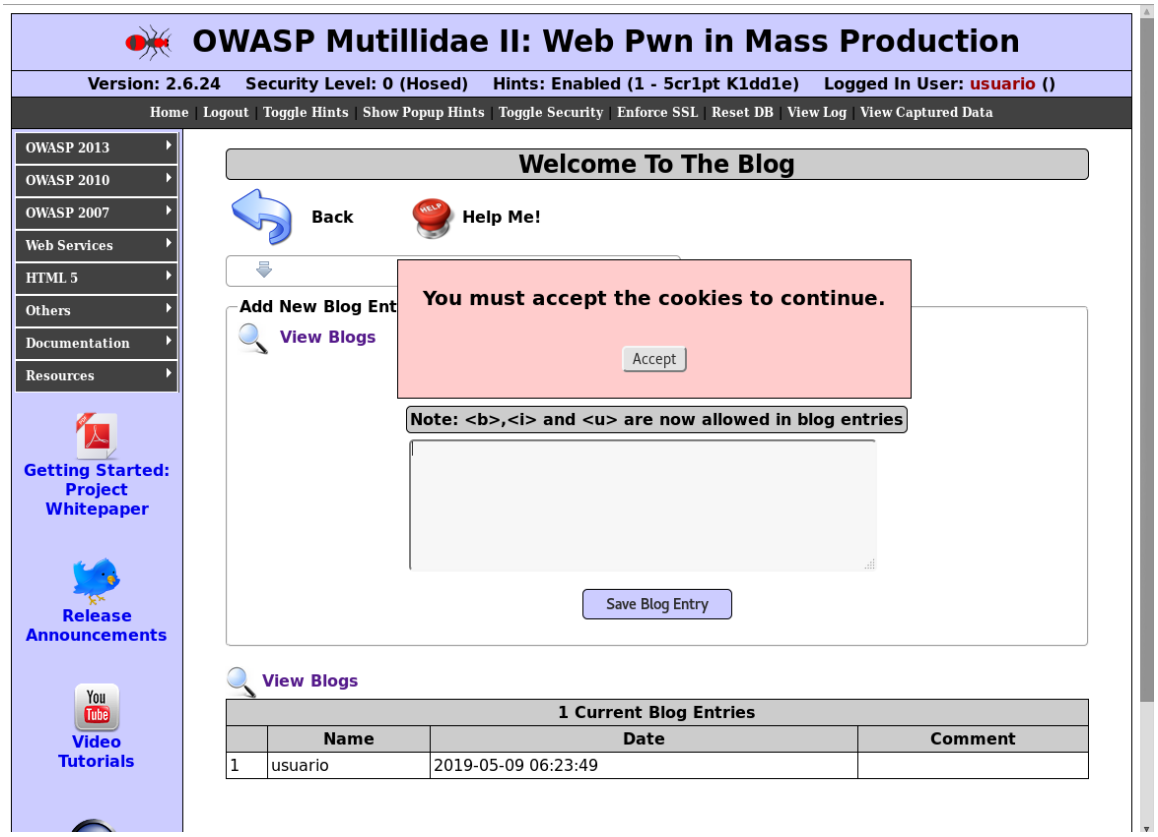
1. Acceso al blog



2. Código del aviso

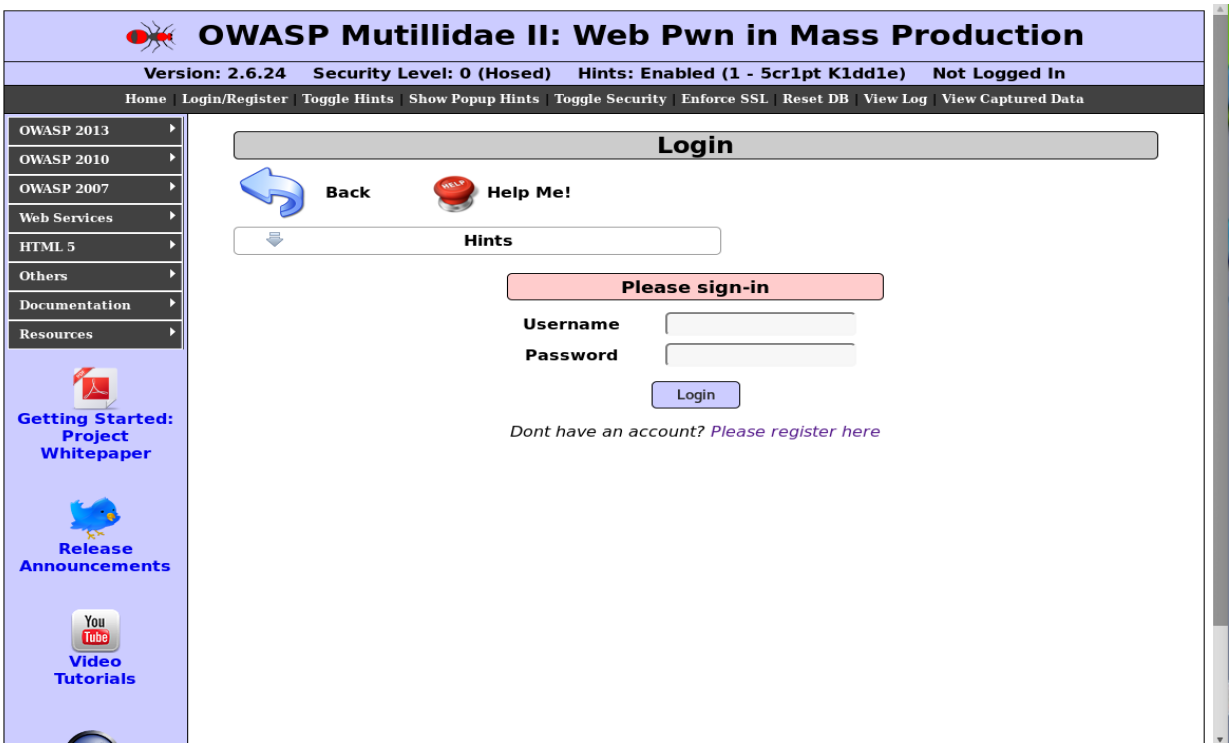


3. Display del aviso



C) Resultado obtenido

4. Cierre de sesión



D) Explicación

Los mensajes del blog se muestran tal y como el usuario los introdujo, sin escapar caracteres especiales. Por lo tanto, podemos inyectar código HTML que será interpretado por el navegador pudiendo introducir formularios o casi cualquier tipo de etiqueta HTML.

En este caso hemos simulado un aviso (introducido en la entrada del blog en la foto 2) que mostrará un mensaje para aceptar cookies (mostrado en la foto 3), el cual suele ser aceptado típicamente sin prestar ningún tipo de atención por la mayoría de los usuarios.

En el código hemos utilizado el atributo *action* del formulario para redirigir al usuario a la página de cierre de sesión (foto 4), lo cual sucede cuando este usuario hace *click* en el botón de "Aceptar".

E) Solución

Antes de guardar el input de la entrada de blog del usuario en la base de datos se debe prestar atención a ese input comprobando si contiene caracteres especiales de HTML, puesto que al mostrarse las entradas del blog directamente, si una de ellas contiene etiquetas HTML, el navegador las interpretará como tal.

Para ello podemos utilizar la función *htmlspecialchars()* de PHP y evitaremos esta inyección muy fácilmente.

F) Opcional (BONUS)

Hemos obtenido el código fuente de Mutillidae II a través de GitHub.

```
G) // Grab inputs
H)             if ($lProtectAgainstSQLInjection){
I)                 // This might prevent SQL injection on the insert.
J)                 $lBlogEntry = $MySQLHandler-
   >escapeDangerousCharacters($_POST["blog_entry"]);
K)             }else{
L)                 $lBlogEntry = $_REQUEST["blog_entry"];
M)             }// end if
```

Estas son las líneas 121 – 127 del archivo *'add-to-your-blog.php'*, en las que se obtiene el texto introducido por el usuario para posteriormente introducirlo en la base de datos mediante esta línea.

```
$SQLQueryHandler->insertBlogRecord($lLoggedInUser, $lBlogEntry);
```

Para evitar la inyección de HTML podríamos cambiar esta última línea por:

```
$SQLQueryHandler->insertBlogRecord($lLoggedInUser,
htmlspecialchars($lBlogEntry));
```

ATAQUE 2

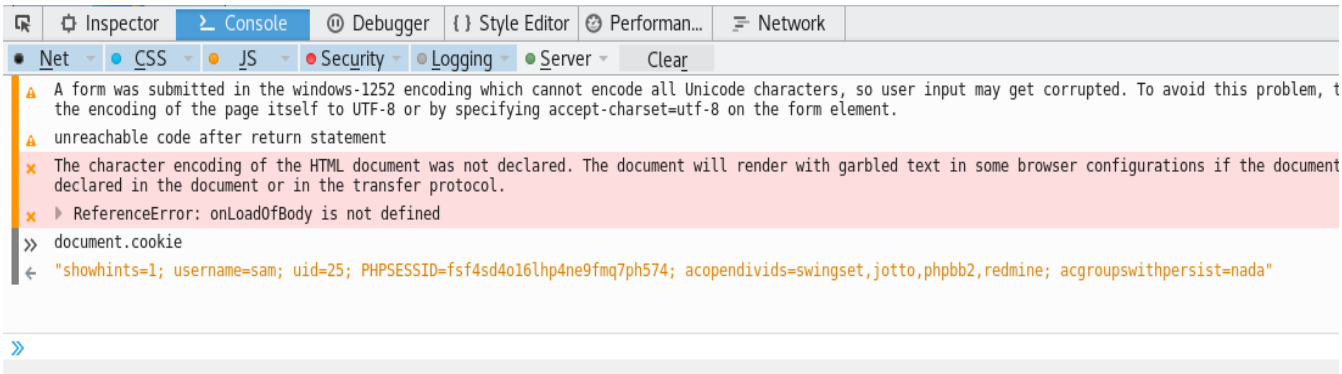
A) Nombre del ataque

A2 – Broken Authentication and Session Management – Authentication Bypass (Via Cookies)

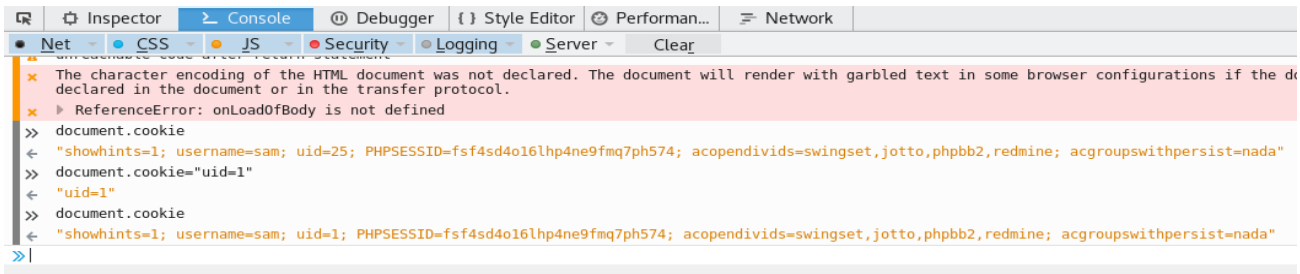
B) Pasos

1. Registro de usuario

2. Visualización de cookies



3. Comando



C) Resultado obtenido

4. Sesión de administrador

D) Explicación

Para que una aplicación web pueda manejar las sesiones de los usuarios que inician sesión, suelen utilizarse las *cookies*, que almacenan la información de la sesión de dicho usuario.

En este caso, al registrarnos (Foto 1) y posteriormente iniciar sesión, hemos podido comprobar la información de nuestras *cookies* al utilizar el comando *document.cookie* en la consola a la que podemos acceder el Mozilla (u otros navegadores como Google Chrome) presionando F12 y accediendo a la pestaña *Console* (Foto 2).

Al utilizar dicho comando hemos podido observar el valor de la *cookie 'uid'*, el cual era 25. Al volver a registrarnos e iniciar sesión con un nuevo usuario, comprobamos que el valor de la misma *cookie* era esta vez 26, es decir, un número más que la anterior.

Con esto podemos deducir que cada vez que se crea un nuevo usuario, el *'uid'* que se le asigna es el número inmediatamente siguiente al *'uid'* anterior. Es por esto que podemos pensar que aquel usuario que tenga el *'uid'* con valor 1, será el primer usuario creado para la página, que comúnmente suele ser el administrador de la página.

De esta manera, al utilizar el comando *document.cookie="uid=1"* en la consola (Foto 3), seremos capaces de editar el valor de nuestra *cookie* y, al refrescar la página podremos comprobar cómo estamos identificados con la cuenta del administrador (Foto 4).

E) Solución

Hay diversas formas de manejar las sesiones de los usuarios que han iniciado sesión en una página web. En este caso se utilizan *tokens* de autorización en el lado del cliente, lo cual no es del todo seguro.

Esa podría ser una forma de evitar este *bypass* de autenticación. Sin embargo, también es posible seguir utilizando *tokens* de autorización

en el lado del cliente y evitar que las *cookies* puedan ser accedidas y editadas por cualquier usuario tan fácilmente.

Para ello, a la hora de crear la *cookie* debemos asignarle la *flag* 'HttpOnly', la cual, si está habilitada, evitaremos que esta *cookie* pueda ser accedida mediante código JavaScript como hemos hecho en la consola de comandos del navegador.

F) Opcional (BONUS)

Hemos obtenido el código fuente de Mutillidae II a través de GitHub.

A la hora de analizar el código, hemos comprobado que la *cookie 'uid'* es establecida mediante las funciones *'setcookie()'* o *'setrawcookie()'* de PHP (líneas 114 y 119 del archivo '*process-login-attemp.php*').

```
if ($lProtectCookies){
        $lUsernameCookie = $Encoder->encodeForURL($lRecord->username);
        setcookie("username", $lUsernameCookie, 0, "", "", FALSE, TRUE);
        setcookie("uid", $lRecord->cid, 0, "", "", FALSE, TRUE);
}
else {
        //setrawcookie() allows for response splitting
        $lUsernameCookie = $lRecord->username;
        setrawcookie("username", $lUsernameCookie);
        setrawcookie("uid", $lRecord->cid);
}// end if $lProtectCookies
```

Para añadirle la *flag* 'HttpOnly' a nuestra cookie es tan sencillo como cambiar el valor del último parámetro de la función a 'TRUE', el cual se corresponde a la activación de dicha *flag*, tal y como se hace en el código anterior en el caso de que la variable *'$lProtectCookies'* valga *True*, lo cual es así en el modo seguro. Como estamos en el modo inseguro, ya que Mutillidae II nos sirve para comprobar las vulnerabilidades de una página web insegura, el valor de esta variable es 'False', por lo que la *cookie* se establece en el apartado *else* del código sin los parámetros opcionales que asignan los valores de *flags* como la de 'HttpOnly', y sus valores son 'FALSE' de forma predeterminada:

```
setcookie ( string $name [, string $value [, int $expire = 0 [, string $path [, string $domain [, bool $secure = false [, bool $httponly = false ]]]]] ) : bool

setrawcookie ( string $name [, string $value [, int $expire = 0 [, string $path [, string $domain [, bool $secure = false [, bool $httponly = false ]]]]] ) : bool
```

Por ello, para añadir la *flag* debemos establecer el valor de la variable *'$lProtectCookies'* a 'True' o cambiar la línea :

```
setrawcookie("uid", $lRecord->cid);
```

a la que se accede en el *else* por:

```
setrawcookie("uid", $lRecord->cid, 0, "", "", FALSE, TRUE);
```

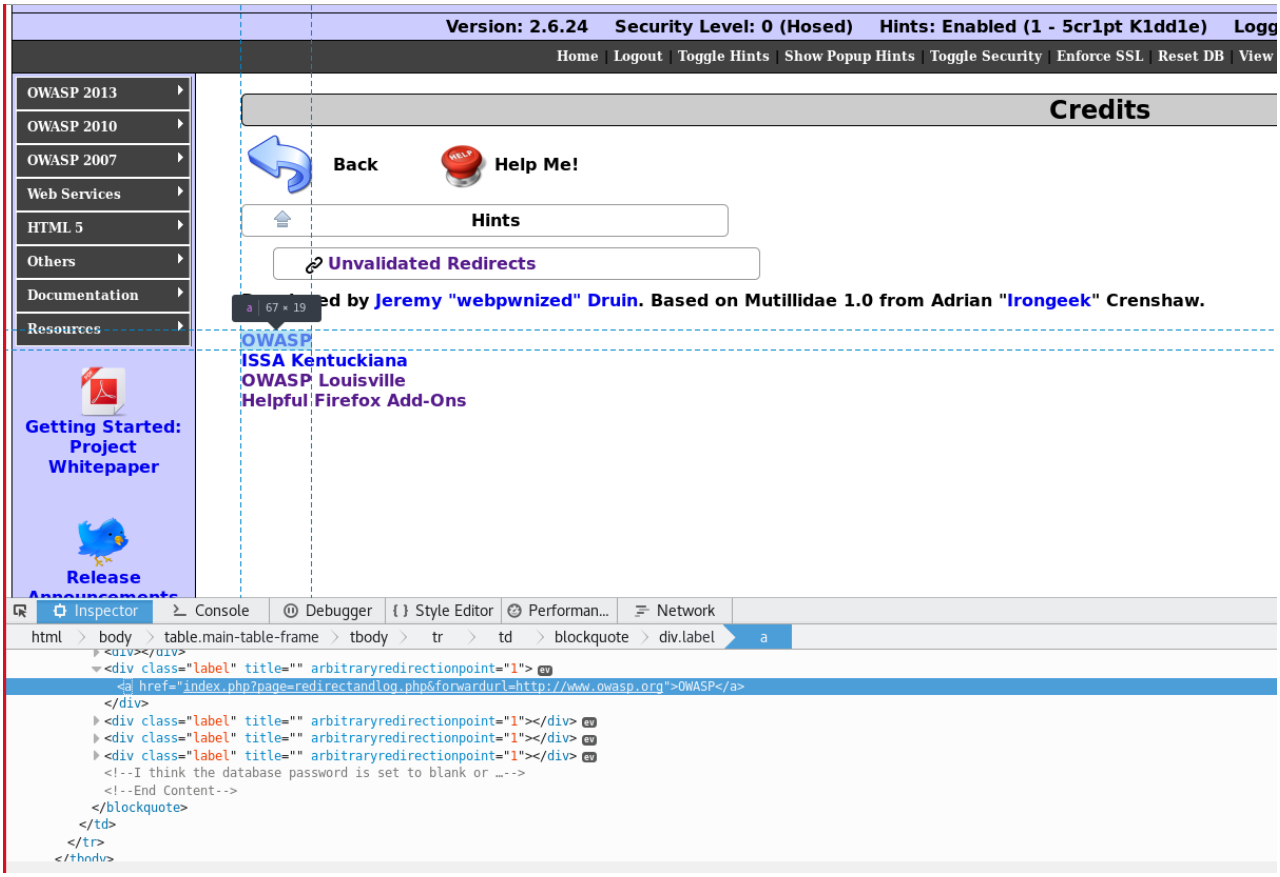
ATAQUE 3

A) Nombre del ataque

A4 – Insecure Direct Object References – Credits – Unvalidated Redirects

B) Pasos

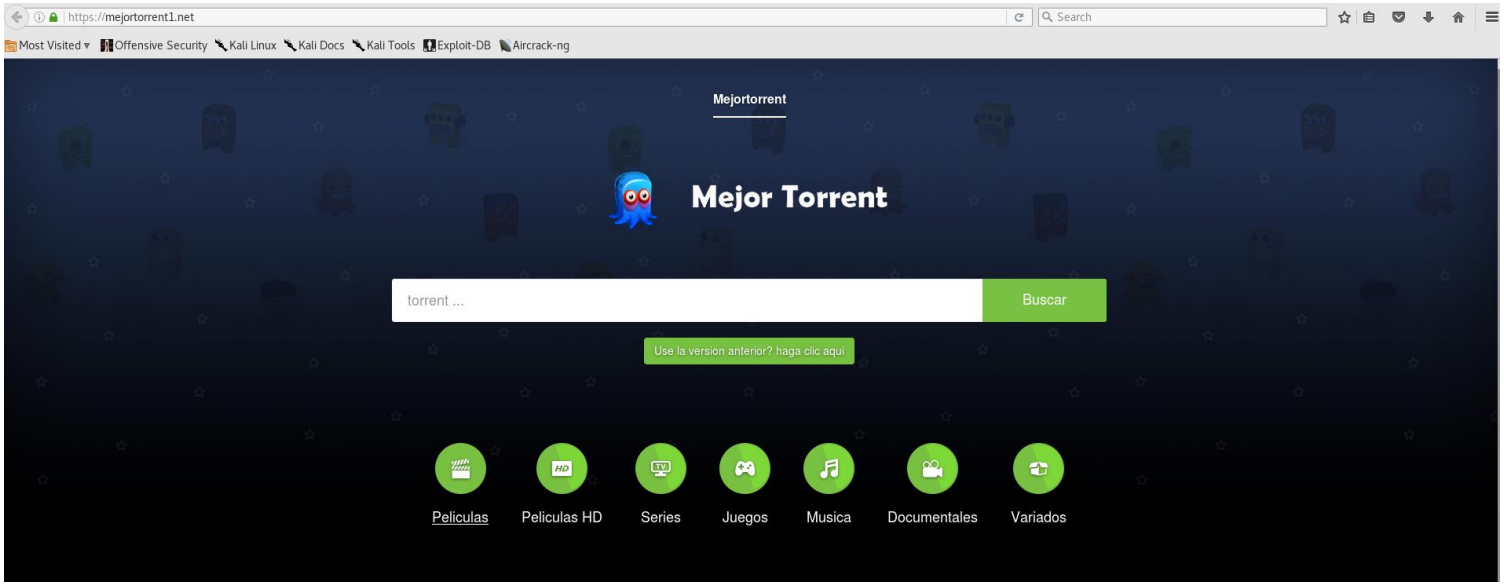
1. Link no validado



2. Link malicioso

C) Resultado obtenido

3. Redirección



D) Explicación

Mutillidae II contiene una página llamada *'Credits'* la cual contiene 4 *links* que redirigen a páginas externas que son de utilidad para los usuarios que quieran informarse más acerca de la página.

Sin embargo, estos *links* redirigen a una página del propio dominio de Mutillidae II (*index.php?page=redirectandlog.php*), la cual se encarga de redirigir una vez más a la página destino establecida en el parámetro GET *'forwardurl'* (Foto 1).

Esta es una vulnerabilidad importante, puesto que cambiando dicho parámetro (Foto 2) podríamos redirigir al usuario a cualquier página que no sea segura (Foto 3), perdiendo así la confianza del usuario que utilizó dicho *link* pensando que el dominio Mutillidae II en este caso, era seguro.

E) Solución

En lugar de establecer el destino del *link* directamente con el parámetro *'forwardurl'* explícitamente, podríamos usar un símbolo como un *integer* para representar el lugar al que queremos ir y de esta manera el *integer* sería comprobado en una especie de tabla en la página web donde cada número corresponda a una URL de destino que de esta manera podemos asegurar que será totalmente segura.

F) Opcional (BONUS)

Hemos obtenido el código fuente de Mutillidae II a través de GitHub.

Como hemos dicho, en los niveles de seguridad 0 y 1, los *links* utilizan este tipo de redirección añadiendo al parámetro *'forwardurl'* directamente la URL de destino:

```
switch ($_SESSION["security-level"]){
        case "0": // This code is insecure
        case "1": // This code is insecure
            $lOWASPURLReference = "http://www.owasp.org";
            $lKYISSAURLReference = "http://www.issa-kentuckiana.org";
            $lOWASPLouisvilleURLReference =
"http://www.owasp.org/index.php/Louisville";
            $lMutillidaeFirefoxAddOnsURLReference =
"https://addons.mozilla.org/en-US/firefox/collections/jdruin/pro-web-
developer-qa-pack/";
```

```
<div class="label" ArbitraryRedirectionPoint="1">
    <a href="index.php?page=redirectandlog.php&forwardurl=<?php echo
$lOWASPURLReference; ?>">OWASP</a>
</div>
<div class="label" ArbitraryRedirectionPoint="1">
    <a href="index.php?page=redirectandlog.php&forwardurl=<?php echo
$lKYISSAURLReference; ?>">ISSA Kentuckiana</a>
</div>
<div class="label" ArbitraryRedirectionPoint="1">
    <a href="index.php?page=redirectandlog.php&forwardurl=<?php echo
$lOWASPLouisvilleURLReference; ?>">OWASP Louisville</a>
</div>
<div class="label" ArbitraryRedirectionPoint="1">
    <a href="index.php?page=redirectandlog.php&forwardurl=<?php echo
$lMutillidaeFirefoxAddOnsURLReference; ?>">Helpful Firefox Add-Ons</a>
</div>
```

Sin embargo, en el nivel de seguridad 5, este parámetro contendrá simplemente un *integer*:

```
case "5": // This code is fairly secure
            $lOWASPURLReference = "2";
            $lKYISSAURLReference = "3";
            $lOWASPLouisvilleURLReference = "4";
            $lMutillidaeFirefoxAddOnsURLReference = "10";
        break;
```

(Archivo *credits.php*)

De esta manera, al redirigirnos a la página *'index.php?page=redirectandlog.php'* y dependiendo del nivel de seguridad, la página nos redirigirá o bien directamente a la URL establecida en el parámetro:

```
switch ($_SESSION["security-level"]){
            case "0": // This code is insecure
            case "1": // This code is insecure
                $forwardurl=$_REQUEST["forwardurl"];
                $LogHandler->writeToLog("Redirected user to: " .
$forwardurl);
                echo '<meta http-equiv="refresh"
content="0;url='.$forwardurl.'">';
                exit
            break;
```

o bien a la URL a la que corresponda el *integer* que se ha pasado por el parámetro *'forwardurl'*:

```
 case "5":
   $forwardurl=$_GET["forwardurl"];
   $lURL = "";
   switch($forwardurl){
       case 1: $lURL = "http://www.irongeek.com/";break;
       case 2: $lURL = "http://www.owasp.org";break;
       case 3: $lURL = "http://www.issa-kentuckiana.org/";break;
       case 4: $lURL = "http://www.owasp.org/index.php/Louisville";break;
       case 5: $lURL = "http://www.pocodoy.com/blog/";break;
       case 6: $lURL = "http://www.room362.com/";break;
       case 7: $lURL = "http://www.isd-podcast.com/";break;
       case 8: $lURL = "http://pauldotcom.com/";break;
       case 9: $lURL = "http://www.php.net/";break;
       case 10:$lURL = "https://addons.mozilla.org/en-
US/firefox/collections/jdruin/pro-web-developer-qa-pack/";break;
   }
   $LogHandler->writeToLog("Redirected user to: " .
$lURL);
   echo '<meta http-equiv="refresh" content="0;url='.$lURL.'">';
    exit;
break;
```

(Archivo *redirectandlog.php*)