



# Seguridad y JWT

PHP



Qualentum Lab

# Seguridad y JWT

La seguridad juega uno de los papeles más importantes en el desarrollo de aplicaciones hoy día. Es un pilar imprescindible para el desarrollo seguro, donde se quiere garantizar la integridad y confidencialidad de los datos. Por este motivo, es necesario dedicarle un fastbook, en el que vamos a explicar cómo realizar los diferentes tipos de configuraciones de seguridad en distintos desarrollos.

¡Empezamos!

*Autor: Jesús Donoso*

- ☰ Introducción a la seguridad en PHP
- ☰ Principios básicos de seguridad en desarrollo web
- ☰ La autenticación de usuarios
- ☰ JSON Web Tokens (JWT)
- ☰ Conclusiones
- ☰ Recursos de interés

# Introducción a la seguridad en PHP



---

La seguridad en el desarrollo web es un elemento imprescindible para garantizar la integridad, confidencialidad y disponibilidad de los datos en las aplicaciones online.

PHP, como uno de los lenguajes de programación más ampliamente utilizados en el ámbito del desarrollo web, proporciona un robusto conjunto de herramientas de seguridad, pero su eficacia se fundamenta en la correcta implementación de los principios sólidos de seguridad.

En este fastbook vamos a exponer los fundamentos clave de la seguridad en PHP, abordando temas que abarcan desde la validación y filtrado de la entrada de usuario hasta la prevención de inyecciones SQL, la protección contra *Cross-Site Scripting* (xss) o la configuración segura del servidor. Recuerda que **la adopción proactiva de buenas prácticas para la seguridad** desde el inicio del desarrollo se traducirá en aplicaciones web fuertes, en la minimización de riesgos y en entornos más confiables.

En los siguientes apartados, por tanto, estudiaremos cómo PHP facilita la implementación de medidas efectivas de seguridad en todas las fases del desarrollo, desde la manipulación segura de datos de entrada hasta la configuración cuidadosa del entorno de ejecución. Al centrarse en una perspectiva proactiva hacia la seguridad, los desarrolladores podemos **crear aplicaciones web no solo eficientes y funcionales**, sino también altamente resistentes frente a las amenazas cada vez más sofisticadas que caracterizan el actual mundo digital. En este sentido, comprender y aplicar los principios fundamentales de seguridad en PHP se convierte en un pilar esencial para cualquier proyecto de desarrollo web exitoso y sostenible.

# Principios básicos de seguridad en desarrollo web



Qualentum Lab

En el desarrollo web, conocer y aplicar los principios básicos de seguridad es prioritario, ya que debemos salvaguardar la integridad y la confidencialidad de la información contenida en nuestra aplicación. Veamos a continuación cuáles son y cómo se llevan a cabo.

- **Validación de entrada.** Todas las entradas de usuario deben ser validadas y filtradas adecuadamente para prevenir ataques de inyección, como la inyección SQL. Por ejemplo, en PHP, se puede utilizar `filter_var()` para validar una dirección de correo electrónico:

```
$email = filter_var($_POST['email'], FILTER_VALIDATE_EMAIL);
```

- **Prevención de inyecciones SQL.** Hay que evitar la construcción manual de consultas SQL. Es mucho más seguro utilizar sentencias preparadas o consultas parametrizadas, por ejemplo:

```
$stmt = $pdo->prepare('SELECT * FROM usuarios WHERE username = :username');
$stmt->bindParam(':username', $username);
$stmt->execute();
```

- **Protección contra Cross-Site Scripting (XSS).** Es necesario ‘escapar’ y filtrar datos antes de mostrarlos en el navegador. En PHP, `htmlspecialchars()` puede ser empleado para evitar la ejecución de scripts maliciosos:

```
echo htmlspecialchars($mensaje);
```

- **Configuración segura del servidor.** Tenemos que asegurarnos de que el servidor está configurado de manera segura. Esto incluye deshabilitar servicios innecesarios y configurar adecuadamente permisos. En el siguiente fragmento de código vemos que se configuran los permisos con el fichero *archivo.php* haciendo uso del comando *chmod*.

```
chmod 644 archivo.php
```

- **Uso de HTTPS.** Como ya sabes, se implementan conexiones seguras mediante HTTPS. Esto se logra mediante la adquisición y configuración de un certificado SSL/TLS para el dominio.
- **Manejo seguro de sesiones.** Tanto el almacenamiento como la gestión de sesiones debe ser segura. Para ello, debemos utilizar identificadores de sesión seguros y renovar las sesiones después del inicio de sesión. Veamos un ejemplo en el que el siguiente fragmento de código PHP utiliza el método `session_start()` para iniciar la sesión en la web de forma segura.

```
session_start(['cookie_secure' => true, 'cookie_httponly' => true]);
```

- **Actualizaciones y parches.** Mantener actualizado el software utilizado en el desarrollo. Aplicar parches de seguridad y actualizaciones regularmente para abordar posibles vulnerabilidades.

- **Principio de menor privilegio.** Esto significa que debemos conceder a los usuarios y procesos solo los privilegios necesarios. Por ejemplo, en bases de datos, es una buena práctica que asignemos unos permisos mínimos requeridos para cada usuario.
- **Auditorías de seguridad periódicas.** Hay herramientas, como OWASP ZAP o Nessus, que podemos utilizarlas para identificar posibles debilidades en el código y la configuración.
- **Educación y concienciación.** Resulta crucial que tanto el equipo humano de desarrollo como los usuarios estén informados sobre cuáles son las mejores prácticas para salvaguardar la seguridad de los entornos y de la información que contienen. Por lo tanto, una buena estrategia es organizar sesiones de formación y proporcionar recursos educativos a todos los implicados.

Ahora que ya tenemos una pista general de las prácticas que debemos llevar a cabo para salvaguardar nuestras aplicaciones, veremos con mayor detalle tres de estos principios básicos: la validación de los datos de entrada, la prevención de inyecciones SQL y la protección contra ataques XSS.

## Validación de datos de entrada

La validación de datos de entrada es un componente crítico en el desarrollo web para garantizar la integridad y seguridad de una aplicación. Este proceso implica verificar y asegurar que los datos proporcionados por el usuario cumplen con ciertos criterios antes de ser procesados. Una validación adecuada previene ataques de inyección, como la inyección SQL y la inyección de código. Por lo tanto, al validar los datos de entrada antes de su procesamiento evitamos que se introduzcan comandos maliciosos en la aplicación. ¿Y con qué pautas se validan los datos de entrada?

¡Vamos a estudiarlas!

### **Pauta 1: garantizar el formato correcto**

Es importante asegurarse de que los datos ingresados cumplen con el formato esperado, especialmente para campos como direcciones de correo electrónico, números de teléfono y fechas.

### **Pauta 2: marcar la longitud y tamaño de datos**

También es necesario establecer una función que verifique que los datos ingresados no exceden límites específicos de longitud o tamaño, previniendo desbordamientos de buffer y mejorando la eficiencia del almacenamiento.

### **Pauta 3: definir el tipo de datos válido de entrada**

Se debe confirmar que los datos ingresados por un usuario son del tipo correcto, por ejemplo: debemos asegurarnos de que un campo numérico realmente contenga un número y no una cadena de texto.

### **Pauta 4: eliminación de caracteres especiales**

En ciertos contextos, puede ser útil eliminar o escapar caracteres especiales que podrían ser utilizados para ataques, como la inyección de scripts (XSS), aunque es importante considerar el contexto específico de uso a la hora de aplicar esta regla.

## Prevención de inyecciones SQL

En el ámbito del desarrollo web, la prevención de inyecciones SQL se destaca como una práctica crítica para salvaguardar las aplicaciones contra ataques maliciosos.

**Este tipo de amenaza ocurre cuando un atacante introduce comandos SQL perjudiciales en las consultas de una aplicación, buscando manipular la base de datos** y, potencialmente, comprometer la integridad de la información almacenada. Para contrarrestar este riesgo, los desarrolladores debemos implementar medidas específicas, por ejemplo:

- **Utilizar sentencias preparadas y parámetros vinculados.**

Este enfoque evita la concatenación directa de valores de usuario en las consultas SQL, asegurando que los datos ingresados no sean interpretados como comandos maliciosos.

```
$nombre = $_POST['nombre'];
$apellido = $_POST['apellido'];

// Uso de sentencias preparadas con PDO
$stmt = $pdo->prepare('INSERT INTO usuarios (nombre, apellido) VALUES (?, ?)');
$stmt->execute([$nombre, $apellido]);
```

- **Uso de consultas parametrizadas.** Al recurrir a consultas parametrizadas, los datos del usuario se proporcionan como parámetros separados de la consulta SQL. De esta manera, se evita que los datos ingresados alteren la estructura de la consulta y se previene la inyección SQL.

```
$nombre = $_POST['nombre'];
$apellido = $_POST['apellido'];

// Uso de consultas parametrizadas con MySQLi
$stmt = $mysqli->prepare('INSERT INTO usuarios (nombre, apellido) VALUES (?, ?)');
$stmt->bind_param('ss', $nombre, $apellido);
$stmt->execute();
```

- **Aplicar el filtrado y la validación de entrada.** Es crucial validar y filtrar meticulosamente la entrada del usuario para asegurar que cumpla con los requisitos esperados antes de incorporar en una consulta SQL. En PHP, las funciones como `filter_var()` son muy útiles para validar y limpiar los datos.

```
$username = $_POST['username'];
// Validación de entrada utilizando filter_var
if (filter_var($username, FILTER_VALIDATE_REGEXP,
array("options"=>array("regexp"=>"/^[\w\W]+$/")))) {
    // El nombre de usuario es válido
    // Continuar con la lógica de la aplicación
} else {
    // El nombre de usuario no cumple con el formato esperado
    // Realizar acciones correspondientes, como mostrar un mensaje de
error
}
```

## Protección contra ataques XSS y CSRF

En el ámbito del desarrollo web, la protección contra ataques XSS (*Cross-Site Scripting*) y CSRF (*Cross-Site Request Forgery*) también desempeña un papel crucial para preservar la seguridad y la integridad de las aplicaciones. Ambos tipos de ataques pueden comprometer la información del usuario y deben ser abordados mediante prácticas específicas. Por eso, en este apartado vamos a ver qué principios debemos aplicar para mitigar estos riesgos, apoyándonos en ejemplos.

## 1

## Protección contra XSS (*Cross-Site Scripting*)

La protección contra XSS constituye un conjunto de estrategias y prácticas destinadas a resguardar las aplicaciones web contra ataques maliciosos.

En la vulnerabilidad XSS, los atacantes insertan scripts dañinos en las páginas web, los cuales se ejecutan en el navegador de los usuarios, comprometiendo la seguridad de la sesión y permitiendo acciones indeseadas. Por lo tanto, debemos tomar dos medidas fundamentales, las que expondremos a continuación, para evitar estos ataques.

- 

**El escapado de datos:** al mostrar datos del usuario en la interfaz, es esencial escapar y filtrar los datos para evitar la ejecución de scripts maliciosos en el navegador del usuario.

```
// Ejemplo de escapado de datos en PHP
$mensaje = $_POST['mensaje'];
echo htmlspecialchars($mensaje);
```

- 

**Content Security Policy (CSP):** implementar políticas de seguridad de contenido (CSP) en los encabezados HTTP ayuda a definir restricciones sobre qué recursos pueden ser cargados y desde dónde. Esto contribuye a prevenir la ejecución de scripts no autorizados.

```
// Ejemplo de implementación de Content Security Policy en PHP
header("Content-Security-Policy: script-src 'self'");
```

## Protección contra CSRF (*Cross-Site Request Forgery*)

La protección contra CSRF es un conjunto de medidas diseñadas para prevenir ataques en aplicaciones web donde un atacante induce a un usuario para que realice acciones no deseadas mientras está autenticado. Estas son las dos más importantes:

- **Token Anti-CSRF:** es decir, comprobar la legitimidad de las solicitudes. Estos tokens deben generarse de manera única para cada sesión.

```
// Ejemplo de generación y verificación de token anti-CSRF en PHP
session_start();
$token = bin2hex(random_bytes(32)); // Generar token único
$_SESSION['csrf_token'] = $token;

// En el formulario HTML
echo '<input type="hidden" name="csrf_token" value="' . $token . '">';

// En el script de procesamiento del formulario
if ($_POST['csrf_token'] === $_SESSION['csrf_token']) {
    // Token válido, procesar la solicitud
} else {
    // Token no válido, posible ataque CSRF
}
```

- **Encabezados HTTP y SameSite Cookies:** hay que configurar encabezados HTTP como 'SameSite' para cookies y utilizar el atributo SameSite como en este ejemplo:

```
// Ejemplo de configuración de SameSite para cookies en PHP
session_start();
session_set_cookie_params(['samesite' => 'Lax']); // O 'Strict' según sea necesario
```

---

**Recuerda que estas medidas deben formar parte integral  
del desarrollo web para asegurar una protección eficaz  
en todos los niveles de tu aplicación.**

En este caso, el código crea un JWT que contiene información sobre un usuario y lo firma con una clave secreta utilizando el algoritmo de hash HMAC SHA256.

### ● Ejemplo de decodificación de un JWT

```
<?php
require_once 'vendor/autoload.php';
use Firebase\JWT\JWT;
// Clave secreta para verificar la firma del token (debería ser la misma
// que la utilizada para la codificación)
$clave_secreta = 'mi_clave_secreta';
// Token a decodificar
$token = "el_token_codificado_a_decodificar";
try {
    // Decodificar el JWT
    $payload_decodificado = JWT::decode($token, $clave_secreta,
array('HS256'));
    // Acceder a la información del payload
    echo "ID de Usuario: " . $payload_decodificado->usuario_id;
    echo "Nombre: " . $payload_decodificado->nombre;
    echo "Rol: " . $payload_decodificado->rol;
} catch (Exception $e) {
    // Manejar errores, por ejemplo, token no válido
    echo "Error: " . $e->getMessage();
}
?>
```

Aquí vemos que el código decodifica un JWT haciendo uso de la misma clave secreta de la codificación. Si el token es válido, accede y utiliza la información contenida en el *payload*. Es fundamental manejar las claves secretas de manera segura y validar los JWT para garantizar la seguridad y la integridad de la autenticación en una aplicación web.

Ahora que ya estamos familiarizados con JWT, vamos a ver cómo se utiliza en **tres frameworks**: Symfony, Laravel y CodeIgniter.

## JWT en Symfony

En Symfony, la implementación de JSON Web Tokens (JWT) es fácil mediante el uso de la biblioteca *lexik/jwt-authentication-bundle*. Para hacer uso de esta biblioteca te recomendamos seguir estos pasos:

Primero instala el paquete *lexik/jwt-authentication-bundle* mediante el administrador de dependencias Composer.

```
composer require lexik/jwt-authentication-bundle
```

A continuación, debes configurar Symfony para integrar la biblioteca. Esto se logra ajustando la configuración del firewall en el archivo *security.yaml*. Es importante incluir el firewall JWT en la configuración.

```
# security.yaml
security:
    firewalls:
        jwt:
            pattern: ^/api
            stateless: true
            anonymous: true
            provider: tu_proveedor_de_usuarios
            guard:
                authenticators:
                    - lexik_jwt_authentication.jwt_token_authenticator
```

El siguiente paso consiste en generar las claves necesarias para firmar los tokens.

Podemos utilizar la herramienta `openssl` para llevar a cabo este proceso:

```
openssl genpkey -out config/jwt/private.pem -aes256 -algorithm rsa -pkeyopt  
rsa_keygen_bits:4096  
openssl pkey -in config/jwt/private.pem -out config/jwt/public.pem -pubout
```

Se debe configurar Symfony para usar las claves generadas mediante la edición del archivo `config/packages/lexik_jwt_authentication.yaml`:

```
# config/packages/lexik_jwt_authentication.yaml  
lexik_jwt_authentication:  
    secret_key:      '%kernel.project_dir%/config/jwt/private.pem' # ruta  
    de la clave privada  
    public_key:     '%kernel.project_dir%/config/jwt/public.pem'  # ruta  
    de la clave pública  
    pass_phrase:   'tu_frase_secreta' # frase secreta para la clave  
    privada  
    token_ttl:      3600
```

Y para generar y validar tokens JWT en controladores Symfony, utilizaremos el servicio `lexik_jwt_authentication.jwt_manager`.

```
use  
Lexik\Bundle\JWTAuthenticationBundle\Services\JWTTokenManagerInterface;  
  
class TuControlador extends AbstractController  
{  
    public function algunaAccion(JWTTokenManagerInterface $jwtManager)  
    {  
        // Generar un token JWT  
        $token = $jwtManager->create($usuario);  
        // Realizar acciones con el token...  
    }  
}
```

La verificación de un token JWT también se puede realizar a través de middleware mediante un Event Listener.

- Primero, crearemos Event Listener que escucha el evento *lexik\_jwt\_authentication.on\_jwt\_authenticated* que se dispara cuando un token JWT ha sido verificado con éxito. Este evento proporciona acceso al token decodificado:

```
// src/EventListener/JwtTokenListener.php
namespace App\EventListener;

use Lexik\Bundle\JWTAuthenticationBundle\Event\JWTAuthenticatedEvent;

class JwtTokenListener
{
    public function onJWTAuthenticated(JWTAuthenticatedEvent $event)
    {
        // Obtener el token decodificado
        $tokenData = $event->getToken()->getPayload();

        // Acciones adicionales según sea necesario
    }
}
```

- El siguiente paso implica registrar este Event Listener como un servicio en el archivo *services.yaml*:

```
# config/services.yaml
services:
    App\EventListener\JwtTokenListener:
        tags:
            - { name: kernel.event_listener, event:
                lexik_jwt_authentication.on_jwt_authenticated, method: onJWTAuthenticated
            }
```

Este servicio estará atento al evento específico y ejecutará la lógica definida en el método *onJWTAuthenticated* cuando se autentique con éxito un token JWT.

Para terminar, me gustaría destacar dos puntos esenciales en esta configuración: en primer lugar, que podemos (y debemos) **adaptar y personalizar** este evento según las necesidades específicas de la aplicación en la que estés trabajando; y, en segundo lugar, que siempre debemos asegurarnos de que la configuración del firewall en el archivo `security.yaml` esté correctamente establecida para proteger las rutas relevantes mediante el firewall JWT.

## JWT en Laravel

En Laravel, el manejo de JWT se simplifica mediante la integración de la biblioteca `tymon/jwt-auth` para la generación y verificación de tokens JWT. Para integrar esta biblioteca y trabajar con JWT en Laravel podemos seguir estos pasos básicos:

- Primero, instalaremos la biblioteca `tymon/jwt-auth` mediante Composer así:

```
composer require tymon/jwt-auth
```

- Tras la instalación, publicaremos las configuraciones predeterminadas ejecutando el comando `artisan`.

```
php artisan vendor:publish --  
provider="Tymon\JWTAuth\Providers\LaravelServiceProvider"
```

Artisan va a crear un archivo de configuración llamado `config/jwt.php`, donde se pueden personalizar las configuraciones según nuestras necesidades.

- Para generar tokens en controladores o en cualquier parte de la aplicación utilizaremos *facades JWTAuth*.

```
use Illuminate\Support\Facades\Auth;
use Tymon\JWTAuth\Facades\JWTAuth;

class AuthController extends Controller
{
    public function login(Request $request)
    {
        // Validar credenciales del usuario
        $credentials = $request->only('email', 'password');
        if (!Auth::attempt($credentials)) {
            return response()->json(['error' => 'Credenciales no válidas'],
401);
        }
        // Generar token JWT
        $token = JWTAuth::fromUser(Auth::user());
        return response()->json(compact('token'));
    }
}
```

- Y para proteger rutas y requerir autenticación mediante JWT, utilizaremos el middleware proporcionado por la biblioteca. Este middleware se agrega en el kernel (*app/Http/Kernel.php*).

```
protected $routeMiddleware = [
    // Otros middlewares...
    'jwt.auth' => \Tymon\JWTAuth\Middleware\GetUserFromToken::class,
    'jwt.verify' => \Tymon\JWTAuth\Middleware\VerifyToken::class,
];
```

- A continuación, podemos aplicar estos middlewares en las rutas correspondientes.

```
Route::group(['middleware' => ['jwt.auth', 'jwt.verify']], function () {
    // Tus rutas protegidas aquí
});
```

- En cuanto a la verificación y decodificación de un token JWT, usaremos el *facades* *JWTAuth*:

```
use Tymon\JWTAuth\Facades\JWTAuth;

class SomeController extends Controller
{
    public function someMethod()
    {
        // Obtener el token desde la solicitud
        $token = JWTAuth::parseToken();

        // Verificar el token
        $user = $token->authenticate();

        // Realizar acciones basadas en el usuario autenticado
    }
}
```

## JWT en CodeIgniter

En CodeIgniter, la implementación de JSON Web Tokens se puede realizar utilizando la biblioteca *firebase/php-jwt*. Para entender y ejecutar esta implementación te compartimos también una guía básica.

- El primer paso será instalar la biblioteca *firebase/php-jwt* utilizando Composer. Por lo tanto, dentro del directorio del proyecto, hay que ejecutar el siguiente comando en la terminal:

```
composer require firebase/php-jwt
```

A continuación, crearemos un controlador o una biblioteca personalizada encargada de manejar la generación y verificación de tokens. En el siguiente fragmento de código compartimos un ejemplo básico de un controlador:

```
// application/controllers/Auth.php

defined('BASEPATH') OR exit('No direct script access allowed');

require APPPATH . 'libraries/REST_Controller.php';
use Restserver\Libraries\REST_Controller;

class Auth extends REST_Controller {

    public function __construct() {
        parent::__construct();
        $this->load->model('user_model'); // Reemplazar 'user_model' con el
        modelo de usuario correspondiente
    }

    public function login_post() {
        $username = $this->post('username');
        $password = $this->post('password');

        // Realizar autenticación (reemplazar este bloque con la lógica de
        autenticación específica)
        $user = $this->user_model->authenticate($username, $password);

        if (!$user) {
            $this->response(['error' => 'Credenciales inválidas'],
            REST_Controller::HTTP_UNAUTHORIZED);
        }

        // Generar un token JWT
        $token = $this->generate_token($user);

        // Devolver el token en la respuesta
        $this->response(['token' => $token]);
    }

    private function generate_token($user) {
        $this->load->library('jwt');

        $payload = [
            'user_id' => $user['id'],
            'username' => $user['username'],
            // Agregar otros claims según sea necesario
        ];

        // Generar el token
        $token = $this->jwt->encode($payload, 'tu_clave_secreta');

        return $token;
    }
}
```

- En los controladores o modelos que requieran protección, se verifica el token antes de permitir el acceso. Para ello, seguiremos este método:

```
// application/controllers/SecureController.php

defined('BASEPATH') OR exit('No direct script access allowed');

require APPPATH . 'libraries/REST_Controller.php';
use Restserver\Libraries\REST_Controller;

class SecureController extends REST_Controller {

    public function __construct() {
        parent::__construct();

        // Verificar el token antes de permitir el acceso
        $this->verify_token();
    }

    private function verify_token() {
        $this->load->library('jwt');

        // Obtener el token desde la solicitud
        $token = $this->input->get_request_header('Authorization');

        if (!$token) {
            $this->response(['error' => 'Token no proporcionado'],
                REST_Controller::HTTP_UNAUTHORIZED);
        }

        try {
            // Decodificar y verificar el token
            $decoded_token = $this->jwt->decode($token,
                'tu_clave_secreta');
        } catch (Exception $e) {
            $this->response(['error' => 'Token inválido'],
                REST_Controller::HTTP_UNAUTHORIZED);
        }
    }

    public function secure_method_get() {
        // Acciones seguras aquí
        $this->response(['message' => 'Acceso autorizado']);
    }
}
```

Estas instrucciones para implementar JWT en CodeIgniter son generales, pero en función de las necesidades y requerimientos de tu aplicación, puedes personalizar el proceso. E insistimos: no olvides proteger adecuadamente la clave secreta y seguir las mejores prácticas de seguridad al implementar la autenticación basada en tokens.

# Conclusiones



---

En este fastbook, hemos hablado de la importancia de la seguridad en el desarrollo web, destacando específicamente la aplicación de JSON Web Tokens (JWT) y otras prácticas fundamentales. Al inicio, hemos revisado los principios básicos de seguridad, tales como la validación de datos de entrada, la protección contra inyecciones SQL y las defensas contra ataques XSS y CSRF. También resultaba clave que entendiésemos la relevancia de la autenticación y autorización en el contexto de la seguridad web.

Una vez que hemos visto los fundamentos teóricos, hemos aprendido a implementar JWT en tres *frameworks* de desarrollo en PHP muy utilizados en la actualidad: Symfony, Laravel y CodeIgniter. Nos hemos apoyado en ejemplos, a modo de guía básica, para entender cómo se instalan y configuran.

Pues bien, no podemos despedir el tema sin recordarte que tomar buenas medidas de seguridad en nuestra web no es un proceso con fecha de finalización, al contrario, debe ser una tarea continua que requiere de una constante revisión. Las amenazas se multiplican cada poco tiempo y los ciberataques se han vuelto día tras día más complejos. De nosotros los desarrolladores depende en gran medida salvaguardar la información de nuestras aplicaciones y de sus usuarios.

# Recursos de interés



Qualentum Lab

Para ampliar tu aprendizaje sobre la seguridad en aplicaciones PHP, te recomendamos que visites la siguiente documentación a través de sus enlaces.

- Symfony JWT:  
<https://symfony.com/bundles/LexikJWTAuthenticationBundle/current/index.html>.
- Laravel JWT: <https://jwt-auth.readthedocs.io/en/develop/>.
- Codeigniter JWT: <https://shield.codeigniter.com/addons/jwt/>.

¡Enhорabuena! Fastbook superado



[Qualentum.com](http://Qualentum.com)