Vigilada Mineducación

Análisis de seguridad de XSS, SQL Injection y CSRF en Laravel, Django, Express y Spring

Analysis of security of XSS, SQL Injection and CSRF in Laravel, Django, Express and Spring

Autor:

Ángel Eduardo Ramos Mena

Tesis de maestría

Asesor(es):

Daniel Correa Botero

Paola Andrea Vallejo Correa

Universidad EAFIT

Maestría en Ingeniería

Medellín – Antioquia

2023

Análisis de seguridad de XSS, SQL Injection y CSRF en Laravel, Django, Express y Spring

Resumen

El desarrollo de aplicaciones tecnológicas ha estado en constante evolución para proveer una mejor experiencia para los usuarios, de igual manera lo ha hecho la capacidad de garantizar la seguridad en ellas, para evitar ciertas amenazas que podrían interferir con su funcionamiento original. A pesar de los esfuerzos, las amenazas a la seguridad, tanto internas como externas, están presentes, razón por la cual es necesario tomar todas las precauciones posibles para responder a ellas. Actualmente, los frameworks de aplicación web (Web Application Frameworks - WAF) facilitan el desarrollo y potencian la seguridad de las aplicaciones web. En este trabajo nos enfocamos en conocer cómo los WAFs *Laravel, Express, Spring y Django,* proporcionan mecanismos para implementar la seguridad en las aplicaciones web. En cada uno de los WAFs seleccionados, se desarrolló una aplicación con el patrón de arquitectura MVC (Modelo - Vista - Controlador). Se seleccionaron las técnicas de hacking *XSS, SQL* Injection y *CSRF,* para alterar las aplicaciones de manera no autorizada. Estas técnicas se usaron para observar cómo se pueden vulnerar las aplicaciones. También se analizó qué tan preparados están los WAFs para hacer frente a dichas técnicas, qué reglas incorporan para garantizar una protección adecuada y cómo se puede minimizar el riesgo para que el desarrollo en un WAF específico sea más seguro.

Palabras Clave: Seguridad**,** Técnica de hacking**,** Modelo-Vista-Controlador (MVC)**,** Framework de aplicación web (WAF), Spring, Laravel, Django, Express, XSS, SQL Injection, CSRF.

Abstract

The development of technological applications has constantly been evolving to provide a better experience for users, as it can ensure their security to avoid specific threats that could interfere with their actual operation. Despite the efforts, internal and external security threats are present, which is why it is necessary to take all possible precautions to respond to them. Currently, web application frameworks (Web Application Frameworks - WAF) facilitate development and enhance security

in web applications. In this work, we focus on how the WAFs Laravel, Express, Spring, and Django, provide mechanisms to implement security in web applications. An application was developed with the MVC (Model - View - Controller) architecture pattern in each of the selected WAFs. Cross-Site Scripting, SQL Injection, and Cross-Site Request Forgery hacking techniques were chosen to alter the applications in an unauthorized manner. These techniques were used to observe how applications can be breached. We also analyzed how prepared WAFs are to deal with these techniques, what rules they incorporate to ensure adequate protection, and how risk can be minimized to make development in a specific WAF more secure.

Keywords: Security**,** Hacking Technique**,** Model-View-Controller (MVC)**,** Web Application Framework (WAF), Spring, Laravel, Django, Express, XSS, SQL Injection, CSRF.

1. Introducción

Los desarrolladores de software son responsables de escribir código que sea óptimo y mantenible, los *frameworks* (conocidos como marcos de trabajo) son una de las herramientas que permite esto, ya que ofrecen varios componentes como librerías, servicios, interfaces, APIs (*Interfaz de programación de aplicaciones*), entre otros, para simplificar la tarea del desarrollo. Los *Web Application Frameworks* (WAF) son una subcategoría de los frameworks que están diseñados para dar soporte al desarrollo de aplicaciones web [1]. Hay diferentes WAFs, cada uno con sus ventajas y desventajas, debido a esto, los desarrolladores deben comprender cómo seleccionar y usar los WAFs para el desarrollo específico que vayan a realizar.

Uno de los factores a considerar al momento de elegir un WAF es la seguridad. Aunque se ha ido mejorando la forma de escribir código (uso de patrones arquitectónicos, código limpio, nombramiento de variables, etc.), también han mejorado los métodos y prácticas para acceder de manera intrusiva a los programas, vulnerando las medidas de seguridad establecidas originalmente, a esos métodos se les conoce como técnicas de hacking [2]. Debido a esto, los desarrolladores deben saber si el WAF seleccionado requiere implementación externa de seguridad, o si ya incorpora algún tipo de configuración para su protección contra las técnicas de hacking.

Este estudio tiene como objetivos:

1. Seleccionar un conjunto de WAFs populares y seleccionar un conjunto de técnicas de hacking comunes.
2. Analizar cómo los WAFs seleccionados protegen contra las técnicas de hacking seleccionadas.
3. Analizar cómo configurar esos WAFs para vulnerarlos ante las técnicas de hacking seleccionadas.

Partimos de una aplicación base para realizar las pruebas de seguridad en cuanto a cada técnica de hacking, con el fin de observar cómo funciona cada una y cómo se puede contrarrestar su efecto.

La estructura de este documento es la siguiente: La sección 2 describe la metodología de investigación. La sección 3 describe la aplicación de las etapas explicadas en la metodología, la cual contiene el enfoque y la pregunta de investigación, la selección de WAFs y técnicas de hacking, la aplicación base y los criterios de comparación. La sección 4 muestra el desarrollo de la aplicación base con su respectiva arquitectura y repositorios. En la sección 5, se aplican las técnicas de hacking en los WAFs, se analizan los resultados, se recopila la información general de los WAFs y se hace la comparación de las técnicas de hacking en los WAFs. En la sección 6 se discuten los resultados de esa comparación. La sección 7 presenta algunos trabajos relacionados con la seguridad de los WAFs. Por último, la sección 8 presenta las conclusiones de esta investigación.

2. Metodología

Este trabajo siguió una guía metodológica basada en el proceso para hacer una investigación cualitativa en contextos de ingeniería, propuesta por Szajnfarber y Gralla [3]. La Figura 1 muestra las etapas de la metodología.

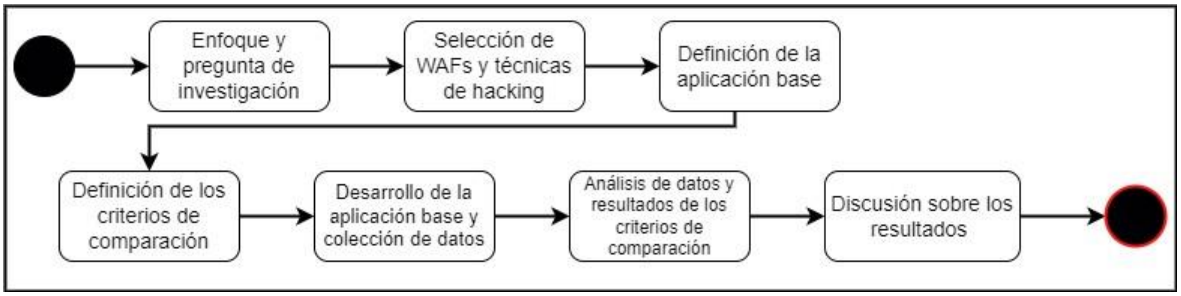

Fig. 1. Etapas de la metodología de investigación. Adaptada de [3].

Primero, definimos el enfoque y la pregunta de investigación (Sección 3.1). Después, seleccionamos los WAFs y las técnicas de hacking a usar (Sección 3.2). Luego definimos una aplicación base con sus componentes (Sección 3.3). Después, definimos una serie de criterios de evaluación para los WAFs y las técnicas de hacking (Sección 3.4). Posteriormente, desarrollamos la aplicación base (Sección 4). Después, probamos las técnicas de hacking en los WAFs, analizamos los resultados, recopilamos la información general de los WAFs y comparamos esas técnicas con base en los criterios de selección (Sección 5). Finalmente, presentamos una discusión de los resultados del estudio (Sección 6).

3. Desarrollo de la metodología

A continuación, vamos a desarrollar cada una de las etapas de la metodología.

3.1. Enfoque y pregunta de investigación

3.1.1. Enfoque de investigación

Para este estudio, nos enfocamos en la selección de los WAFs más populares que implementan (o permiten implementar) el patrón MVC o similares, para determinar qué clase de protección tienen frente a algunas técnicas de hacking. Nos interesamos en MVC dado que "MVC ha sido ampliamente adaptado para el diseño e implementación de sistemas web. Hoy en día muchos frameworks permiten el desarrollo de sistemas web utilizando este estilo, por ejemplo, Spring, Django, Laravel y Yii." [4].

3.1.2. Pregunta de investigación

¿Qué mecanismos y/o medidas de protección ofrecen algunos de los WAFs más populares contra algunas de las técnicas de hacking más comunes?

3.2. Selección de WAFs y técnicas de hacking

3.2.1. WAFs

La selección de WAFs se hizo con base en dos fuentes de información: (i) GitHub Octoverse y (ii) Repositorios WAFs de GitHub.

GitHub Octoverse. Es un reporte anual que muestra las tendencias en GitHub. El reporte muestra información acerca de la popularidad de los lenguajes de programación y los proyectos open-source. En la sección "*Top languages over the years*", se indica que los lenguajes de programación más populares en 2021 fueron (i) JavaScript, (ii) Python, (iii) Java, (iv) Typescript, (v) C#, (vi) PHP, (vii) C++, (viii) Shell, (ix) C, y (x) Ruby [5], de esos lenguajes seleccionamos los seis primeros.

Repositorios WAF de GitHub. La mayoría de los WAFs alojan su código en repositorios de GitHub. Esos repositorios pueden ser comparados con respecto a dos elementos: (i) **Estrellas (Stars):** Las estrellas en un repositorio actúan como un marcador y una apreciación del proyecto. Muchas veces, la puntuación de estrellas se asemeja a la calidad que tiene el proyecto. (ii) **Bifurcaciones (Forks):** Significa crear una copia de un proyecto. Los desarrolladores usan el fork para trabajar en nuevas características, luego hacen un *pull request* que contiene los cambios hechos al proyecto original. Para cada lenguaje, seleccionamos el WAFs más popular, medido por estrellas y forks, que permita implementar el patrón MVC o similares.

La Tabla 1 muestra el número de estrellas y forks que los usuarios dieron a los repositorios de los WAFs alojados en GitHub (información verificada el 05/12/2022). El WAF con la puntuación más alta es Laravel (71.6k estrellas), seguido por Django (67.6k estrellas), Spring Boot (64.5k estrellas), Express (59.1k estrellas) y por último ASP NET Core (30.3k estrellas). De esos cinco WAFs seleccionamos los cuatro que tuvieran más estrellas y forks. Es importante destacar que Express está basado en NodeJS. Para JavaScript/TypeScript, buscamos frameworks que soporten NodeJS, ya que este es requerido para que pueda procesar la información hacia el back-end.

Tabla 1. Lista de WAFs más populares.

WAF	Lenguaje	Estrellas (Stars)	Bifurcaciones (Forks)	Enlace de GitHub
Laravel	PHP	71.6k	23.2k	https://github.com/laravel/laravel
Django	Python	67.6k	28.3k	https://github.com/django/django
Spring Boot	Java	64.5k	37.6k	https://github.com/spring-projects/spring-boot
Express	NodeJS (JavaScript/ TypeScript)	59.1k	10k	https://github.com/expressjs/express
ASP NET Core	C#	30.3k	8.5k	https://github.com/dotnet/aspnetcore

A continuación, se dará una breve explicación sobre los WAFs seleccionados:

Laravel: Laravel es un marco de aplicación web PHP con una sintaxis expresiva y elegante. Laravel es gratuito y fue creado en 2011. El propósito principal de Laravel es facilitar la creación de código de una manera simple, habilitando múltiples funcionalidades utilizando las características de las versiones más recientes de PHP. Las características de Laravel incluyen el motor de plantillas Blade, compatibilidad con MVC, un ORM llamado Eloquent, solicitudes fluidas y soporte de caché [6].

Django: Django es un marco web de alto nivel de Python que fomenta un desarrollo rápido y un diseño limpio y pragmático. Es gratuito y de código abierto y facilita la creación de mejores aplicaciones web de forma rápida y con menos código. Django es rápido, seguro y escalable, además permite implementar aplicaciones siguiendo el patrón de diseño MVC [7].

Spring Boot: Spring es un marco basado en Spring para desarrollar aplicaciones web en el lenguaje de programación Java. Spring se basa en el patrón de inversión de dependencias, el cual permite construir sistemas altamente desacoplados. Spring consta de un marco MVC, un marco de validación y un control transaccional de bases de datos. Spring Boot facilita la creación de aplicaciones basadas en Spring con una configuración mínima. Las funciones de Spring Boot incluyen: la incorporación de Tomcat, Jetty o Undertow sin implementar archivos WAR y proporciona dependencias de inicio obstinadas para simplificar la configuración de compilación [8].

Express: Express es un marco web minimalista, rápido y de código abierto para Node.js. Proporciona una robusta configuración para el desarrollo de aplicaciones web y móviles, con funciones como enrutamiento de URL, opciones para administrar sesiones y cookies, facilidades para motores de plantillas y buena cobertura de pruebas. También cuenta con una gran comunidad y manuales que apoyarán cualquier proyecto con Express [9].

Es importante mencionar que los WAFs pueden ser clasificados en opinionados y no opinionados. Los WAFs opinionados tienden a incluir la mayoría de las partes que se necesitan para crear una aplicación. Principalmente, puede ser autenticación incorporada, un enrutador y posiblemente una capa de base de datos [10]. Por ejemplo, Laravel es un WAF opinionado. Laravel define una estructura y una arquitectura del proyecto, contiene muchas librerías y ayudas para la gestión de la base de datos, autenticación, sesión web, entre otras. La ventaja es que un desarrollador puede implementar aplicaciones web MVC muy rápidamente. Sin embargo, el rendimiento puede verse afectado.

Por el otro lado, están los WAFs no opinionados, que vienen con las mínimas funcionalidades. Muchos de ellos no definen una arquitectura y estructura de un proyecto. Aunque el rendimiento aumenta, un desarrollador web debe tomar muchas decisiones críticas, como definir la arquitectura de la aplicación. Además, el desarrollador debe incluir librerías de terceros, como una librería para conectar y administrar la base de datos. Express es un ejemplo de un WAF no opinionado de NodeJS.

Finalmente, muchos WAFs son considerados minimalistas, los cuales vienen con mínimas funcionalidades y librerías. Muchos WAFs no opinionados (como Express) también son minimalistas.

Hemos definido conceptos interesantes sobre los WAFs, ya que se usarán para definir las características generales de los WAF en los siguientes capítulos.

3.2.2. Técnicas de hacking

La selección se hizo con base en algunas de las técnicas de hacking más críticas, según OWASP (Open Web Application Security Project). De acuerdo con la fundación OWASP, algunas de las técnicas de hacking más críticas incluyen XSS, CSRF y SQL Injection [11]. Seleccionamos estas tres técnicas y las describimos a continuación:

XSS (Cross-Site Scripting): Es un tipo de técnica de hacking que usa scripts maliciosos para ser inyectados en las páginas web. La Figura 2 muestra, con un ejemplo, el funcionamiento de la técnica XSS [12]:

1) Un atacante (hacker) inyecta un script malicioso en un sitio web de confianza (trusted website).
2) La víctima visita ese sitio web y acciona el script malicioso.
3) El navegador web de la víctima, ejecuta ese script y sin darse cuenta, reenvía la información deseada (cookies, token de sesión, etc.) al atacante.

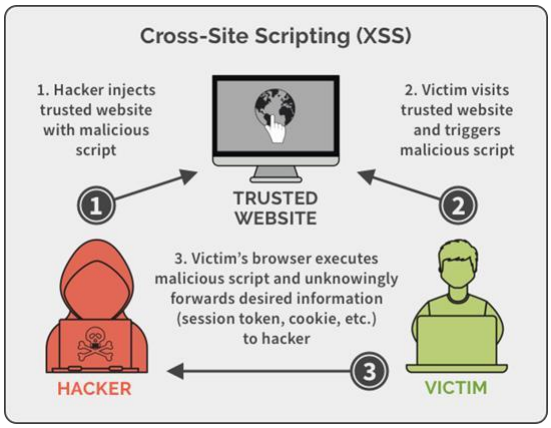

Fig. 2. Ataque XSS. Tomado de [12].

Los ataques XSS pueden ser categorizados como: Almacenados, Reflejados y basados en DOM (*Document Object Model*).

- **XSS Almacenado:** "Es el tipo de ataque en el que los scripts maliciosos quedan alojados en los servidores objetivos, como bases de datos, foros, campos de comentarios, etc. El script se vuelve a ejecutar, cuando la víctima vuelve a hacer la petición a la página sobre la información almacenada en el servidor. Este ataque también puede ser referido como persistente o XSS tipo-I" [13].
- **XSS Reflejado:** "Es aquel ataque en el que la secuencia de comandos inyectada se refleja en el servidor web, como en un mensaje de error, resultado de búsqueda o cualquier otra respuesta que incluya parte o la totalidad de la entrada enviada al servidor como parte de la solicitud. Este ataque también puede ser referido como no persistente o XSS tipo-II" [13].
- **XSS Basado en DOM: "**Este tipo de ataque, que es menos conocido, ejecuta una carga útil del ataque, dando como resultado la modificación del "entorno"

DOM (Document Object Model) en el navegador de la víctima, utilizando script por el lado del cliente original, esta vulnerabilidad se presenta en el lado del cliente. Este ataque también puede ser referido como XSS tipo-0" [14].

SQL Injection: Es un tipo de ataque que consiste en la inyección de comandos SQL a través de la entrada de los datos [15]. "Las aplicaciones web a menudo utilizan formularios y entradas de datos de URLs, con el fin de crear sentencias SQL que tienen como objetivo obtener o escribir desde una base de datos. La inyección SQL consiste en manipular esas formas, las cuales se envían los datos para que puedan cambiar la semántica de las sentencias SQL" [16].

La Figura 3 muestra, con un ejemplo, el funcionamiento de la técnica de hacking SQL Injection [17]:

1) Un atacante (hacker) detecta un sitio web vulnerable basado en SQL e ingresa una consulta SQL maliciosa a través de los datos de entrada (website input fields).
2) La consulta SQL es validada y el comando es ejecutado por la base de datos.
3) El atacante obtiene accesos para ver o alterar registros, o puede actuar como administrador de la base de datos (database).

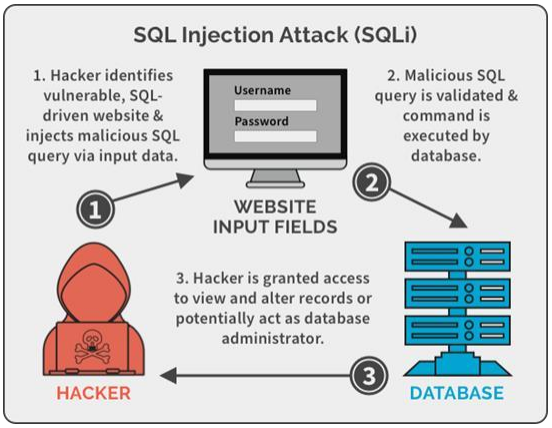

Fig. 3. Ataque SQL Injection. Tomado de [17].

Hay otros tipos de ataques relacionados a SQL, como:

- **Blind SQL Injection**: "Es un tipo de ataque de inyección SQL, que hace consultas verdaderas o falsas a la base de datos y determina la respuesta en función de la respuesta de las aplicaciones. Este ataque se usa a menudo cuando la aplicación web está configurada para mostrar mensajes de error genéricos, pero no ha mitigado el código que es vulnerable a la inyección SQL" [18].
- **ORM Injection:** Es un ataque usando SQL Injection contra un modelo de objeto de acceso a datos generado por un ORM. Desde el punto de vista de un tester (encargado de verificar la funcionalidad de un programa), este ataque es

visualmente idéntico al ataque de SQL Injection, sin embargo, la vulnerabilidad de Inyección existe en el código generado por la capa ORM [19].

- **Double Encoding:** "Este ataque consiste en codificar dos veces los parámetros de la solicitud de un usuario en formato hexadecimal, para eludir los controles de seguridad o causar comportamientos inesperados en la aplicación. Es posible porque el servidor web acepta y procesa la solicitud del cliente en muchas formas codificadas" [20].
- **Code Injection:** "Es un término general para los tipos de ataques que consisten en inyectar código, que luego es interpretado/ejecutado por la aplicación. Este tipo de ataque aprovecha el deficiente manejo de los datos que no son de confianza, generalmente tiene éxito debido a la falta de una validación adecuada de los datos de entrada/salida, por ejemplo:
    - caracteres permitidos (clases de expresiones regulares estándar o personalizadas)
    - formato de datos
    - cantidad de datos esperados" [21].

CSRF (Cross Site Request Forgery): "Es un tipo de técnica de hacking que engaña a la víctima para que envíe una solicitud maliciosa, hereda la identidad y los privilegios de la víctima, para realizar una función no deseada en nombre de ésta" [22]. "Este ataque tiene éxito debido a que los desarrolladores tienden a confiar en el hecho de que un cliente, nunca enviará una solicitud a la que no esté vinculado o, uno, que la GUI (Interfaz Gráfica de usuario) no está diseñada para enviar. A diferencia de los ataques XSS, que explotan la confianza del cliente en el sitio web, este ataque explota la confianza del sitio web en el cliente" [16].

La Figura 4 muestra, con un ejemplo, el funcionamiento de la técnica de hacking CSRF [23]:

1) La víctima (victim) inicia sesión en su cuenta de un sitio web bancario.
2) El sitio web del banco (bank website) asigna a la víctima un token de validación.
3) El atacante (hacker) envía una solicitud falsificada disfrazada de una comunicación legítima del banco.
4) La víctima sin saberlo reenvía la solicitud al banco.
5) La solicitud falsificada es ejecutada por el banco usando el token de validación previamente asignado.

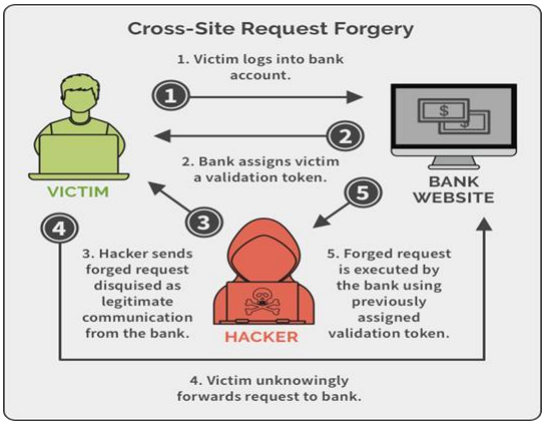

Fig. 4. Ataque CSRF. Tomado de [23].

3.3. Definición de la aplicación base

Para este estudio decidimos desarrollar una aplicación web tipo blog llamada *Blog Posts*, está aplicación contiene un formulario que recibe un título y una descripción dada por el usuario (ambos campos son obligatorios), los datos enviados quedan guardados en la base de datos como una publicación (*post*), y, por consiguiente, el usuario podrá entrar a esa publicación (*post*), agregar comentarios (*comments*) y borrarlos.

Esta aplicación sigue el patrón de arquitectura MVC y se compone de las siguientes partes:

- **Aplicación Web:** Esta aplicación será desplegada en un navegador. Para su desarrollo se usó HTML, CSS y JavaScript.
- **Publicaciones y comentarios:** La aplicación permitirá el manejo de publicaciones (*posts)* y comentarios (*comments*). Podrá listar las publicaciones (*posts)*, crearlas, mostrar una publicación (*post)* con sus comentarios (*comments)*, crear comentarios (*comments)* y borrarlos.
- **Base de datos MySQL:** La aplicación usará una base de datos MySQL para almacenar información de las publicaciones (*posts*) y comentarios (*comments)*.
- **Bootstrap CSS:** La aplicación será diseñada con Bootstrap, un framework gratuito y de código abierto de CSS dirigido al desarrollo web responsivo.

Arquitectura de la aplicación

La aplicación web se basa en dos partes para funcionar: parte del cliente y parte del servidor. La parte del cliente, conocida como *Frontend*, es la parte que interactúa con el usuario (la parte visual). El Frontend está desarrollado en HTML, CSS, y JavaScript. La parte del servidor, conocida como *Backend*, proporciona la información al Frontend (por ejemplo, la información de las publicaciones (*posts*) y/o comentarios (*comments*)). Los frameworks MVC, permiten a los desarrolladores

desarrollar tanto el código del Frontend como del Backend en una sola aplicación. Entonces, los usuarios finales o clientes se pueden comunicar con el servidor a través de peticiones HTTP, lo que permite al usuario navegar a través de las páginas web. Para la comunicación con las bases de datos, se usaron los ORMs (Mapeo Objeto-Relacional) El ORM específico depende del WAF usado en el desarrollo de la aplicación.

La Figura 5 muestra el diagrama de clases de la aplicación Blog Posts, el cual incluye los atributos que posee cada clase y cómo es su relación. Una publicación (*post*) contiene un id, un título, una descripción, las fechas de creación y actualización y puede tener uno o varios comentarios (*comments*). Un comentario (*comment*) está relacionado a una publicación (*post*) y contiene un id, un mensaje y las fechas de creación y actualización.

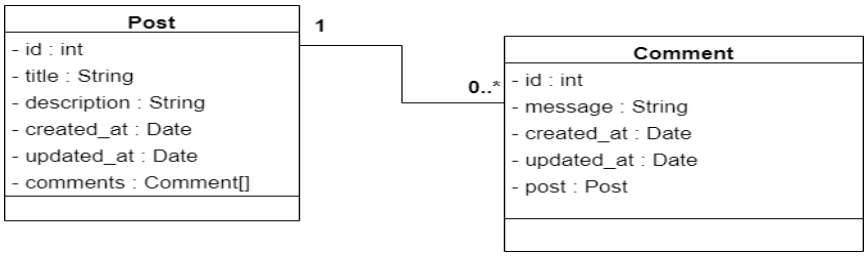

Fig. 5. Diagrama de clases de la aplicación Blog Posts.

3.4. Criterios de evaluación

En esta sección se define la información general a recopilar de los WAFs seleccionados (Tabla 2) y, los criterios de comparación de los mecanismos que poseen (o soportan) los WAFs contra las técnicas de hacking (Tabla 3).

Tabla 2. Información general de los WAFs seleccionados.

ID	Característica	Descripción
GF1	Lenguaje de programación	Especifica el lenguaje de programación soportado por el WAF
GF2	Opinionado	Especifica si el WAF es opinionado o no
GF3	Minimalista	Especifica si el WAF es minimalista o no
GF4	Última versión y fecha de lanzamiento	Especifica la última versión estable y la fecha de lanzamiento del WAF
GF5	Patrón de arquitectura	Especifica si el WAF sugiere el uso de un patrón de arquitectura de software específico

Tabla 3. Criterios de comparación de los mecanismos de protección en los WAFs ante las técnicas de hacking.

ID	Criterio	Descripción
GH1	Componente interno de protección (si aplica)	Especifica si el WAF incorpora por defecto un componente para la protección contra XSS o CSRF o SQL Injection, según aplique*. Si posee alguno, se nombra ese componente
GH2	Componente externo de protección (si aplica)	Si el WAF no incorpora un componente interno, entonces se especifica si el WAF requiere algún componente externo para la protección contra XSS o CSRF o SQL Injection, según aplique
GH3	Uso del componente de protección	Especifica cómo usar el componente de protección interno o externo (según aplique) en el WAF contra XSS o CSRF o SQL Injection, según aplique
GH4	Uso obligatorio del componente de protección	Especifica si al utilizar el componente (interno o externo), es obligatorio implementar algún mecanismo de protección contra XSS o CSRF o SQL Injection, según aplique, para ejecutar correctamente la aplicación
GH5	Modificación de la aplicación para hacerla vulnerable ante la técnica de hacking	Especifica qué configuraciones o cambios se hacen en la aplicación para que sea vulnerable ante XSS o CSRF o SQL Injection, según aplique
GH6	Documentación del componente	Especifica el enlace donde se encuentra la documentación del componente (interno o externo) relacionada con la protección contra XSS o CSRF o SQL Injection, según aplique

Esta tabla se evaluará por cada técnica de hacking seleccionada

4. Desarrollo de la aplicación Blog Posts y colección de datos

Para la creación de la aplicación Blog Posts en los WAFs seleccionados, definimos la siguiente estructura:

- **Controllers:** Se definieron dos controladores para esta aplicación. (i) **HomeController**, el cual se encarga del manejo de la interacción del usuario con el *home page* y el *about page*. (ii) **PostController**, el cual se encarga del manejo de la interacción del usuario con la lista de publicaciones (*posts*), la visualización de una publicación (*post*) específica con sus comentarios

(*comments)*, la creación de una nueva publicación (*post*), y la creación y eliminación de comentarios (*comments*) dentro de una publicación (*post*) específica.

- **Models:** Se definieron dos modelos. (i) **Post** con sus correspondientes atributos y relaciones (basados en los elementos definidos en el diagrama de clases de la Figura 5). (ii) **Comment** con sus correspondientes atributos y relaciones (basados en los elementos definidos en el diagrama de clases de la Figura 5).
- **Views:** Se definieron cinco vistas con las siguientes carpetas. (i) Carpeta **layouts** que contiene la vista ***master*** (contiene la estructura del estilo principal del sitio web). (ii) Carpeta **home** que contiene las vistas del home y del about us. (iii) Carpeta **post** que contiene la vista de todos las publicaciones (*posts*) y de los comentarios (*comments*) de cada publicación (*post*).

Es importante resaltar que la documentación oficial de Django sugiere usar el patrón de arquitectura MVT (Modelo-Vista-Template). Debido a que el patrón es similar al patrón MVC, decidimos usar MVC en Django para facilitar la comparación.

Blogs desarrollados

A continuación, se listan los enlaces en GitHub de la aplicación Blog Posts desarrollada en los cuatro WAFs seleccionados. Se crearon dos repositorios para cada WAF, un repositorio en el que se aplica la protección contra las técnicas de hacking seleccionadas y otro repositorio en el que no se aplica la protección contra las técnicas de hacking seleccionadas:

- **Laravel con protección:**
  https://github.com/FrameworksSecurity/LaravelWithProtection
- **Laravel sin protección:**
  https://github.com/FrameworksSecurity/LaravelWithoutProtection
- **Django con protección:**
  https://github.com/FrameworksSecurity/DjangoWithProtection
- **Django sin protección:**
  https://github.com/FrameworksSecurity/DjangoWithoutProtection
- **Express con protección:**
  https://github.com/FrameworksSecurity/ExpressWithProtection
- **Express sin protección:**
  https://github.com/FrameworksSecurity/ExpressWithoutProtection
- **Spring con protección:**
  https://github.com/FrameworksSecurity/SpringWithProtection

- **Spring sin protección:**
  https://github.com/FrameworksSecurity/SpringWithoutProtection

La Figura 6 presenta la arquitectura de la aplicación Blog Posts en los WAFs seleccionados, la cual incluye la estructura descrita anteriormente y cómo está compuesta dependiendo del WAF.

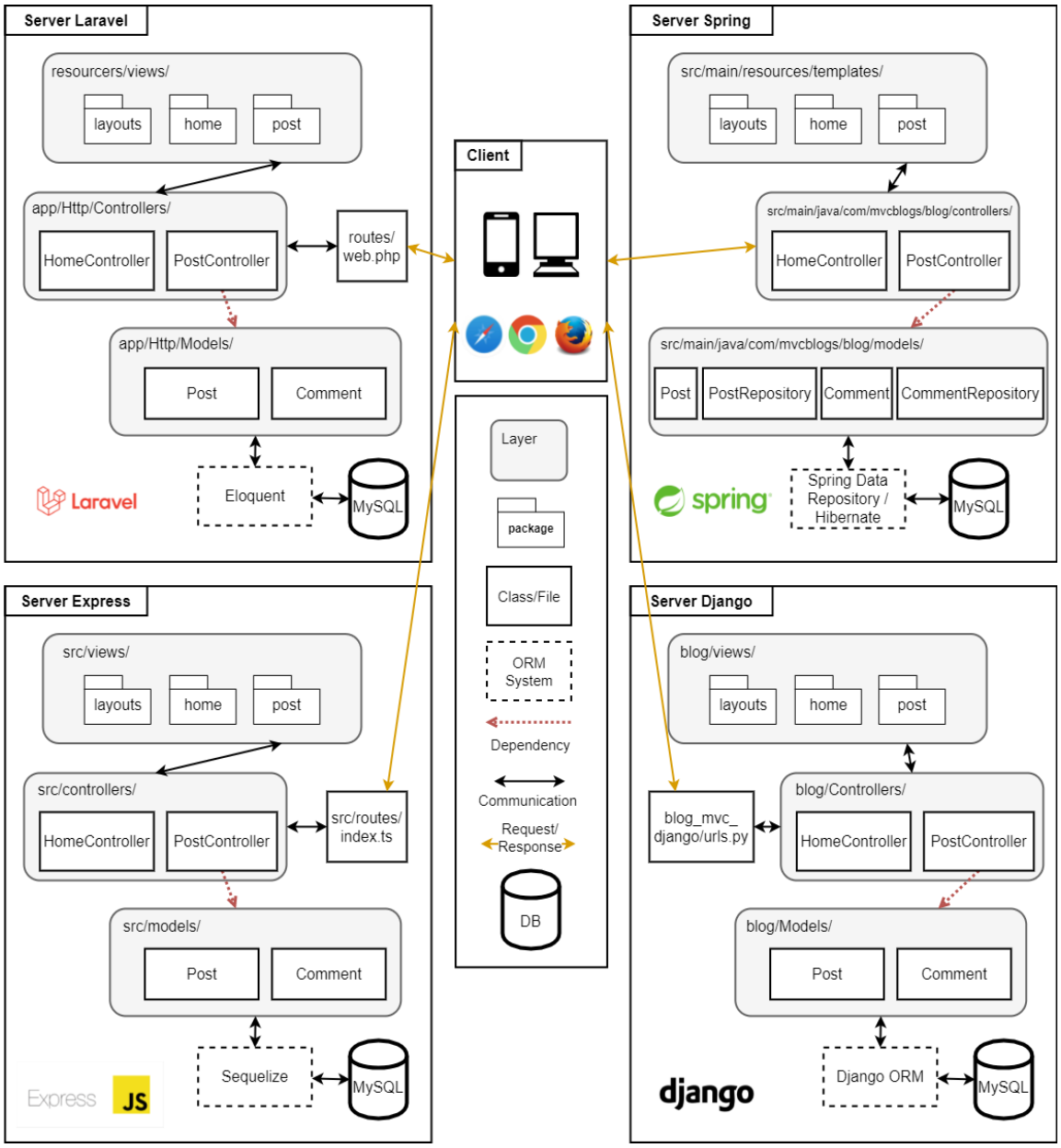

Fig. 6. Arquitecturas MVC de la aplicación Blog Posts en Laravel, Django, Spring y Express.

La Figura 7 muestra la aplicación desarrollada en los 4 WAFs, desplegada en el navegador web.

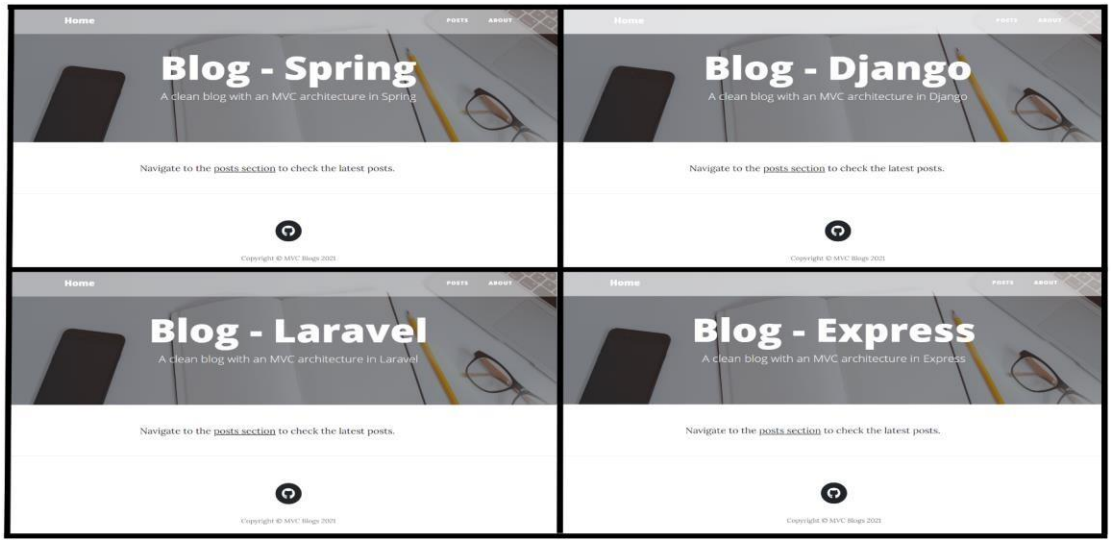

Fig. 7. Pantalla de inicio de la aplicación Blog Posts en ejecución en los WAFs seleccionados.

La Figura 8 muestra el formulario de una de las cuatro aplicaciones web para crear publicaciones (*posts*).

Fig. 8. Formulario para la creación de una nueva publicación (*post*).

La Figura 9 muestra una lista de publicaciones (*posts*) creadas en una de las cuatro aplicaciones web.

Fig. 9. Información agregada a la aplicación Blog Posts.

5. Análisis de datos y resultados de los criterios de comparación

El objetivo de esta sección es analizar cómo los WAFs seleccionados proveen mecanismos de protección (o no) contra las técnicas de hacking seleccionadas. Es decir, si tienen instrucciones incorporadas para la protección, documentación, escritura de código específica, etc.

Para este análisis, se usará como base las aplicaciones Blog Posts desarrolladas en los cuatro WAFs (ver sección 4), y se harán las respectivas pruebas de seguridad, luego, se harán las comparaciones sobre las técnicas de hacking seleccionadas (ver sección 3.2). Esto nos permitirá saber cómo funciona la aplicación en cada WAF con protección y sin protección frente a las técnicas que se están analizando.

Antes de presentar los resultados, veamos cómo aplicamos cada técnica de hacking para saber si la aplicación base estaba protegida o no.

5.1. Aplicación de las técnicas de hacking

5.1.1. XSS

Para aplicar XSS, seguimos los siguientes pasos:

1) Tomamos una de las aplicaciones base sin protección (en este caso, Laravel). Y accedimos a la siguiente ruta:

   http://localhost:8000/posts

2) En la ruta anterior se encuentra el formulario de la aplicación Blog Posts, con los campos *Post Title* y *Post Description.* En el campo *Post Title* se puede escribir cualquier texto (por ejemplo, “Test”). En el campo *Post Description* escribiremos la siguiente línea:

   ```
   <script>alert("Hi, Lack of Security");</script>
   ```

   La línea anterior ejecutará un script con un cuadro de alerta con el mensaje `"Hi, Lack of Security"`. En la Figura 10 se muestra cómo diligenciar el formulario de la aplicación Blog Posts para esta prueba.

Fig. 10. Agregando información para el funcionamiento de la técnica Cross-Site Scripting.

3) Hacemos clic en “Submit” para guardar la información anterior.

   Al hacerlo, aparecerá un cuadro de alerta con el mensaje `"Hi, Lack of Security"` (ver Figura 11).

Fig. 11. Ejecución del script ingresado en la figura 10.

4) Hacemos clic en “Ok” y nos regresará a la vista del formulario de la aplicación (ver Figura 12).

Fig. 12. Resultados de la ejecución del script ingresado en la figura 10.

Podemos ver que se guardó el título que ingresamos anteriormente, pero la descripción quedó vacía. A esto se le conoce como **Stored XSS**, traducido como "**XSS Almacenado**", debido a que cada vez que se ejecute la ruta de la aplicación donde está guardada la información, también lo hará ese script que se mandó, porque está guardado en la base de datos como un script y no como un texto plano.

Para saber si aplicación Blog Posts tiene protección contra XSS, tomamos una de las aplicaciones base con protección (en este caso, Laravel), repetimos los primeros tres pasos anteriores y este fue el resultado (ver Figura 13).

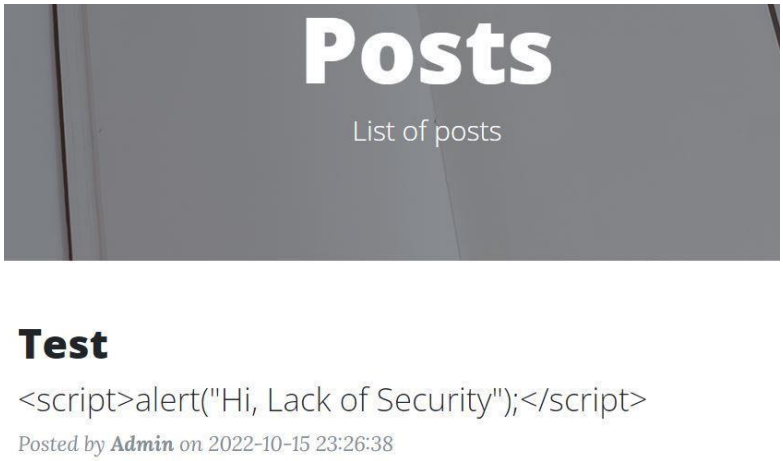

Fig. 13. Ejecución fallida del script ingresado en la figura 10.

La aplicación guardó la información correctamente, de esa forma, cada vez que accedamos a esa ruta, no se ejecutará el script que se mandó en el campo *Post Description,* debido a que se guardó como texto plano.

5.1.2. SQL Injection

Para aplicar SQL Injection seguimos los siguientes pasos:

1) Tomamos una de las aplicaciones base sin protección (en este caso, Laravel). Y accedimos a la siguiente ruta:

   http://localhost:8000/posts

2) En la ruta anterior se encuentra el formulario de la aplicación Blog Posts, con los campos *Post Title* y *Post Description*. En el *Post Title* se puede escribir cualquier texto (por ejemplo, "Test"). En el campo *Post Description* escribiremos la siguiente línea (ver Figura 14).

```
'); ALTER TABLE posts RENAME hacked; (SELECT * FROM posts where '1'='1
```

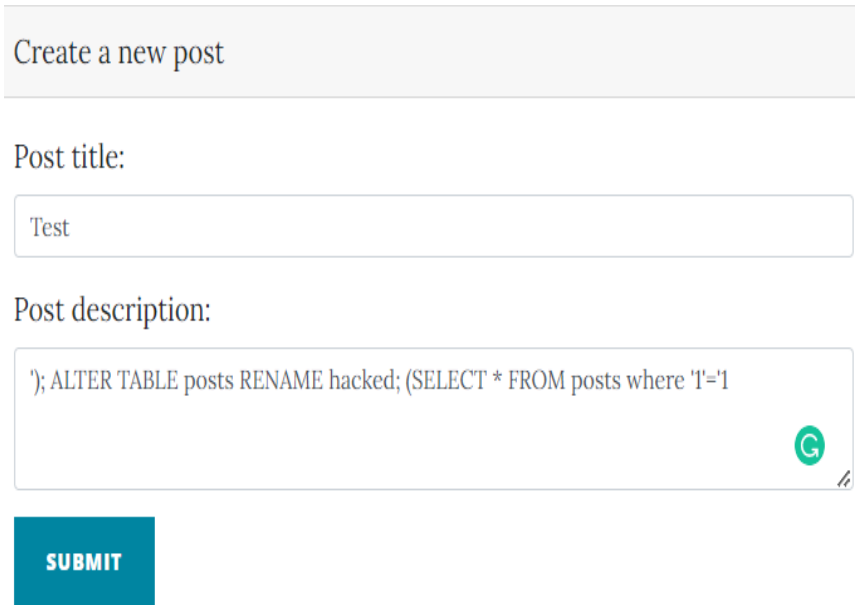

Fig. 14. Agregando información para el funcionamiento de la técnica SQL Injection.

La sentencia que está programada en la aplicación Blog Posts para guardar información en la base de datos es la siguiente:

```
INSERT INTO posts (title,description) VALUES ('title value', 'description value');
```

La sentencia anterior guarda la información en la base de datos a través del formulario, en la figura 14, se observa el formulario de la aplicación Blog Posts con la información diligenciada.

Con la información del campo Post Description, estamos alterando la sentencia programada de la siguiente manera:

```
INSERT INTO posts (title,description) VALUES ('Test',''); ALTER TABLE posts RENAME hacked; (SELECT * FROM posts where '1'='1');
```

Si la aplicación no está protegida, en lugar de guardar esa información como texto plano, se tomará como una consulta SQL con múltiples sentencias (cada sentencia está delimitada por ;). Lo que significa que se ejecutarán tres sentencias:

(i) `INSERT INTO`, que inserta información en la tabla posts, específicamente en las columnas title y description. La inserción se hace a través de `VALUES`, que especifica cuál es la información que se guardará en esas columnas, en este

caso, se guardará la información que venga desde los campos del formulario de la aplicación Blog Posts.

(ii) `ALTER TABLE`, que modificará la tabla existente ***`posts`***, a través de `RENAME`, que cambiará el nombre de esa tabla a ***`hacked`***.

(iii) `SELECT * FROM`, que traerá todos los datos de la tabla ***`posts`*** a través de `where`, el cual es usado como filtro para traer los datos de esa tabla, siempre y cuando se cumpla una condición específica, en este caso, se debe cumplir que `'1'` sea igual a `'1'`.

3) Hacemos clic en “Submit” para guardar la información anterior. Al hacerlo, aparecerá este tipo de error (ver Figura 15).

Fig. 15. Ejecución exitosa de las sentencias SQL ingresadas en la figura 14.

En la figura 15, se puede observar que la aplicación dejó de funcionar, porque no encuentra el nombre de la tabla llamada ***`posts`***, ya que ahora esa tabla se llama ***`hacked`***, a causa de la sentencia SQL `ALTER TABLE`. Esto es posible porque la aplicación está configurada para ejecutar sentencias directamente en la base de datos, por ende, la información enviada se completó de forma que sea una consulta válida para la sintaxis de SQL.

Al ataque anterior se le conoce como **Blind SQL Injection**, traducido como “**Ataque a ciegas por inyección SQL**”. Ya que el atacante, al enterarse de que el campo *Post Description* reconoce sintaxis SQL, puede experimentar con inyecciones para saber cómo está estructurada esa base de datos, hasta que obtenga información de su interés, ya sea el nombre de la base de datos, los registros que hay allí, e incluso modificar su estructura.

Para saber si aplicación Blog Posts tiene protección contra SQL Injection, tomamos una de las aplicaciones base con protección (en este caso, Laravel), repetimos los pasos anteriores y este fue el resultado (ver Figura 16).

Fig. 16. Ejecución fallida de la sentencias SQL ingresadas en la figura 14.

La aplicación guardó la información correctamente sin permitir que se modificara la estructura de la base de datos.

5.1.3. CSRF

Para aplicar CSRF, seguimos los siguientes pasos:

1) Usamos un programa externo que nos permite realizar pruebas API llamado *Postman,* tomamos la ruta que está programada para guardar los datos en la aplicación Blog Posts (en este caso, Django):

   http://localhost:8000/posts/save

2) En *Postman*, para mandar una petición usamos el formato JSON (Notación de Objecto de JavaScript), el cual se estructura de la siguiente manera para el envío de información a la aplicación Blog Posts:

```
{
    "title":"Article",
    "description":"The best article of the world"
}
```

   La estructura anterior indica que el campo *title* tendrá el valor de `"Article"` y el campo *description* tendrá el valor de `"The best article of the world"`. Haciendo referencia a cómo mandamos la información en el formulario de la aplicación Blog Posts.

3) Hacemos clic en “Send” y este fue el resultado (ver Figura 17).

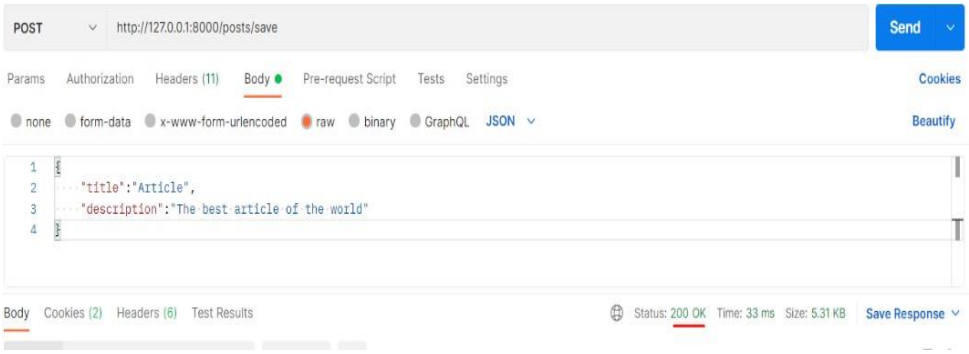

Fig. 17. Petición exitosa hacia la aplicación Blog Posts desde *Postman*.

Si al mandar la información a través de *Postman* se muestra un código de estado `200`, significa que la petición fue exitosa, lo que indica que quien hizo la petición se ha hecho pasar por el cliente (en este caso *admin*), que estaba activo en la aplicación web y esta no tuvo manera de comprobar si era el usuario legítimo quien envió esa información.

4) Verificamos que la información enviada desde Postman, se vea reflejada en la aplicación (ver Figura 18).

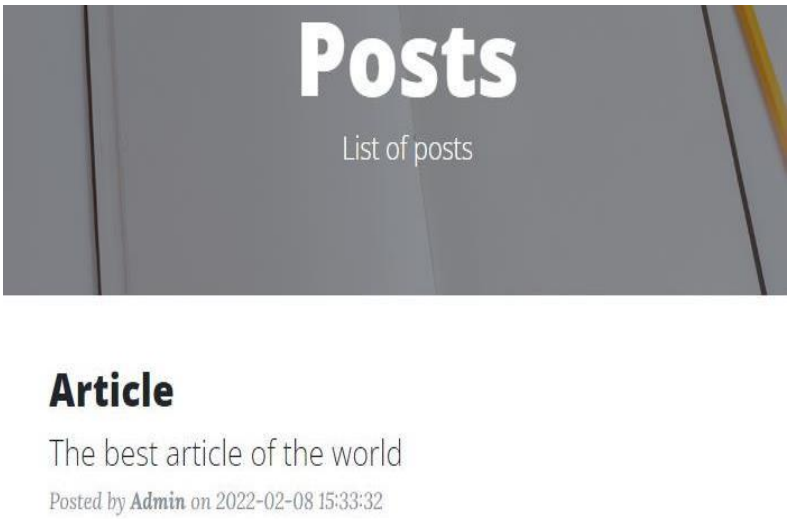

Fig. 18. Información agregada en la aplicación Blog Posts desde *Postman*.

Para saber si aplicación Blog Posts tiene protección contra CSRF, tomamos una de las aplicaciones base con protección (en este caso, Django), repetimos los pasos anteriores y este fue el resultado (ver Figura 19).

Fig. 19. Petición denegada hacia la aplicación Blog Posts.

Si el código de estado que aparece después de enviar la petición es un `403`, o algún otro que se asemeje al significado de este, como el `419`, indica que la petición fue denegada debido a que no tiene el token, que es lo que actúa como único identificador en esa sesión del usuario para permitir el envío de la información. Indicando que la aplicación solo recibe información desde el formulario, lo que significa que la aplicación está protegida.

5.2. Características generales de los WAFs

En la tabla 3 presentamos los resultados de las características generales de los WAFs, estos resultados fueron verificados el 05/12/2022. Recopilamos la información desde las referencias presentadas en la sección 3.2 y la sección 3.4.

Tabla 3. Resultados sobre las características generales de los WAFs.

ID	Laravel	Django	Express	Spring
GF1	PHP	Python	NodeJS (Javascript/TypeScript)	Java
GF2	Sí	Sí	No	Sí
GF3	No	No	Sí	No
GF4	9.3.12 (22-noviembre-2022)	4.1.3 (01-noviembre-2022)	4.18.2 (8-octubre-2022)	3.0.0 (24-noviembre-2022)
GF5	Arquitectura MVC	Arquitectura MVT	No obliga a usar un único patrón de arquitectura. Express genera rutas simples y un patrón de vistas	Arquitectura por capas o limpia

5.3. Resultados de la comparación de las técnicas de hacking en los WAFs

Luego de desarrollar las 4 aplicaciones base y de aplicar el procedimiento definido en la sección 5.1, a continuación, presentamos los resultados de la comparación de las técnicas de hacking en los WAFs seleccionados.

En las tablas 4, 5 y 6 presentamos la comparación general sobre los mecanismos de protección en los WAFs frente a las técnicas de hacking. Estos resultados fueron verificados el 17/09/2022. La información fue recopilada basada en las referencias de la sección 3.2 y los criterios de la sección 3.4.

Tabla 4. Resultados de la comparación de los mecanismos de protección en los WAFs respecto a la técnica XSS (Cross-Site Scripting).

ID	Laravel	Django	Express	Spring
GH1	Sí - Laravel incluye un motor de plantillas llamado Blade que contiene elementos de protección	Sí - Django provee su propio motor de plantillas llamado DTL (Django Template Language) que contiene elementos de protección	No	No
GH2	No	No	Sí - Se utilizó el motor de plantillas EJS (Embebbed JavaScript templating) que contiene elementos de protección	Sí - Se utilizó el motor de plantillas ThymeLeaf que contiene elementos de protección
GH3	Los ataques XSS en Laravel comúnmente se dan en la capa visual. El motor de plantillas Blade, utiliza la sentencia *{{}}* para desplegar información sensible en las vistas (al interior de esta sentencia se coloca el nombre de las variables que vienen desde el controlador y se desean proteger contra XSS). Esa información sujeta a ser desplegada puede ser vulnerable a ataques XSS. Por lo tanto, la sentencia *{{}}* escapea (hace que la información que se está enviando sea tratada como información y no de otra	Los ataques XSS en Django comúnmente se dan en la capa visual. El motor de plantillas DTL, utiliza la sentencia *{{}}* para desplegar información sensible en las vistas (al interior de esta sentencia se coloca el nombre de las variables que vienen desde el controlador y se desea proteger contra XSS). Esa información sujeta a ser desplegada puede ser vulnerable a ataques XSS. Por lo tanto, la sentencia *{{}}* escapea (hace que la información que se está enviando sea tratada como	Los ataques XSS en Express comúnmente se dan en la capa visual. El motor de plantillas EJS, utiliza la etiqueta <%=%> para desplegar información sensible en las vistas (al interior de esta sentencia se coloca el nombre de las variables que vienen desde el controlador y se desean proteger contra XSS). Esa información sujeta a ser desplegada puede ser vulnerable a ataques XSS. Por lo tanto, la etiqueta <%=%> escapea (hace que la información que se está enviando sea tratada como	Los ataques XSS en Spring comúnmente se dan en la capa visual. El motor de plantillas ThymeLeaf, utiliza el atributo *th:text* para desplegar información sensible en las vistas (al interior de esta sentencia se coloca el nombre de las variables que vienen desde el controlador y se desean proteger contra XSS). Esa información sujeta a ser desplegada puede ser vulnerable a ataques XSS. Por lo tanto, el atributo *th:text* escapea (hace que la información

	manera, por ejemplo, como un script) la información que contiene para eliminar caracteres extraños que puedan contener ataques XSS	información y no de otra manera, por ejemplo, como un script) la información que contiene para eliminar caracteres extraños que puedan llevar ataques XSS	información y no de otra manera, por ejemplo, como un script) la información que contiene para eliminar caracteres extraños que puedan llevar ataques XSS	que se está enviando sea tratada como información y no de otra manera, por ejemplo, como un script) la información que contiene para eliminar caracteres extraños que puedan llevar ataques XSS
GH4	No	No	No	No
GH5	En lugar de utilizar la sentencia {{ }} para mostrar el contenido de variables que se pasan internamente (y que protege contra XSS), utilizamos la sentencia *{!! !!},* la cual evita la protección contra ataques XSS	En lugar de utilizar la sentencia {{ }} para mostrar el contenido de variables que se pasan internamente (y que protege contra XSS), utilizamos la sentencia *{{ | safe}},* la cual evita la protección contra ataques XSS	En lugar de utilizar la etiqueta <%=%> para mostrar el contenido de variables que se pasan internamente (y que protege contra XSS), utilizamos la etiqueta *<%-%>,* la cual evita la protección contra ataques XSS	En lugar de utilizar el atributo th:text para mostrar el contenido de variables que se pasan internamente (y que protege contra XSS), utilizamos el atributo *th:utext,* el cual evita la protección contra ataques XSS
GH6	https://laravel.com/docs/9.x/blade#displaying-data	https://docs.djangoproject.com/en/4.1/topics/security/#cross-site-scripting-xss-protection	https://ejs.co/ - (Sección Tags). https://dl.acm.org/doi/pdf/10.1145/3184558.3188736 - (Sección 2.1)	https://www.thymeleaf.org/doc/tutorials/2.1/usingthymeleaf.html#unescaped-text

Tabla 5. Resultados de la comparación de los mecanismos de protección en los WAFs respecto a la técnica SQL Injection.

ID	Laravel	Django	Express	Spring
GH1	Sí - Laravel incluye el ORM llamado Eloquent que contiene elementos de protección	Sí - Django tiene un ORM incluido llamado Django' ORM que contiene elementos de protección	No	No
GH2	No	No	Sí - Se utilizó el ORM Sequelize que contiene elementos de protección	Sí - Se utilizó el ORM Spring Data Repository que contiene elementos de protección
GH3	Los ataques SQL Injection en Laravel se dan cuando se realizan operaciones sobre la base de datos. Si el programador utiliza Eloquent para llevar a cabo cualquier operación sobre base de datos, automáticamente estará protegido contra SQL Injection	Los ataques SQL Injection en Django se dan cuando se realizan operaciones sobre la base de datos. Si el programador utiliza Django ORM para llevar a cabo cualquier operación sobre base de datos, automáticamente estará protegido contra SQL Injection	Los ataques SQL Injection en Express se dan cuando se realizan operaciones sobre la base de datos. Si el programador utiliza Sequelize para llevar a cabo cualquier operación sobre base de datos, automáticamente estará protegido contra SQL Injection	Los ataques SQL Injection en Spring se dan cuando se realizan operaciones sobre la base de datos. Si el programador utiliza Spring Data Repository para llevar a cabo cualquier operación sobre base de datos, automáticamente estará protegido contra SQL Injection
GH4	No - pero si se utiliza el ORM por defecto (Eloquent) si	No - pero si se utiliza el ORM por defecto (Django ORM) si	No	No
GH5	En lugar de utilizar el sistema ORM que provee Laravel, utilizamos el paquete Facade que provee métodos para hacer operaciones con bases de datos, usamos los métodos	En lugar de utilizar el sistema ORM que provee Django, utilizamos la librería *mysql.connector* para establecer la conexión con la base de datos, luego usamos el método *cursor.execute(),* para	Sequelize puede ser vulnerable si se usa el método *Sequelize.query()*, se debe configurar la conexión a la Base datos permitiendo que se pueda ejecutar una consulta con múltiples declaraciones a la	En lugar de utilizar el sistema ORM Spring Data Repository, utilizamos la librería *Java.sql.Connection* para hacer la conexión con la base de datos, utilizamos los métodos *createStatement()* para crear el objeto que se va a

	DB::unprepared y *DB::raw* de ese paquete para ejecutar sentencias SQL en la base de datos, lo cual evita la protección contra ataques SQL Injection	ejecutar sentencias SQL directamente en la base de datos, se habilitó la capacidad de ejecutar una consulta con múltiples declaraciones, escribiendo dentro de ese método la característica *multi=true* después de la instrucción a ejecutar, la cual evita la protección contra ataques SQL Injection	vez con la característica *multipleStatements:true*, la cual evita la protección contra ataques SQL Injection	mandar a la base de datos y *executeUpdate()* para ejecutar dicho objeto, también se configuró la conexión a la base de datos, para ejecutar una consulta con múltiples declaraciones a la vez, con la característica *allowMultiQueries=true* en las propiedades de la aplicación, la cual evita la protección contra ataques SQL Injection
GH6	https://www.enrichment.iocspublisher.org/index.php/enrichment/article/view/346 - Abstract	https://docs.djangoproject.com/en/4.1/topics/security/#sql-injection-protection. https://scholar.googleusercontent.com/scholar?q=cache:xe-i-UG0t-UJ:scholar.google.com/+Django+ORM+sql+injection&hl=es&as_sdt=0,5 - (Introduction)	https://oa.upm.es/48283/8/TFM_JORGE_MARTINEZ_LASCORZ.pdf - Sección 4.4	https://www.digitalocean.com/community/tutorials/sql-injection-in-java - Best Practices to avoid SQL Injection

Tabla 6. Resultados de la comparación de los mecanismos de protección en los WAFs respecto a la técnica CSRF (Cross Site Request Forgery).

ID	Laravel	Django	Express	Spring
GH1	Sí - Laravel viene con un middleware llamado VerifyCsrfToken que contiene elementos de protección	Sí - Django viene con un middleware llamado CsrfViewMiddleware que contiene elementos de protección	No	No
GH2	No	No	Sí - Se utilizó la librería Csruf que contiene elementos de protección	Sí - Se utilizó la dependencia org.springframework.security que contiene elementos de protección
GH3	Los ataques CSRF en Laravel se dan cuando se ejecutan comandos maliciosos haciéndose pasar por el usuario. Laravel automáticamente genera un token (el cual valida que el usuario autenticado sea quien está haciendo realmente la petición) para cada sesión de usuario activa. En cualquier formulario HTML que se vaya a hacer una petición, Laravel utiliza la sentencia que provee la directiva Blade *@csrf* para validarla, la cual debe ir dentro del formulario.	Los ataques CSRF en Django se dan cuando se ejecutan comandos maliciosos haciéndose pasar por el usuario. Django automáticamente genera un token (el cual valida que el usuario autenticado sea quien está haciendo realmente la petición) para cada sesión de usuario activa. En cualquier formulario HTML que se vaya a hacer una petición, Django utiliza la sentencia *{%csrf_token%}* para validarla, la cual debe ir dentro del formulario.	Nosotros usamos la librería *csurf* y procedimos a adecuarla en las rutas del proyecto, una vez hecho esto se debe mandar el token a la vista del proyecto a través de un input tipo oculto con su nombre y respectivo valor que se le pasa desde el controlador *<input type="hidden" name="csrf" value="<%=csrfToken%> ">*, el cual debe ir dentro del formulario.	Para hacer uso de la protección se deben hacer algunas configuraciones. Nosotros agregamos la librería *org.springframework.security* y la protección contra CSRF de esa librería de forma manual en las dependencias, agregamos la configuración para esa librería en una clase y finalmente le pasamos el token a la vista como un input de tipo oculto *<input type="hidden" th:name="${_csrf.parameterName}" th:value="${_csrf.token}" />*.
GH4	Sí - De lo contrario, la aplicación responderá a la petición con un código de	Sí - De lo contrario, la aplicación responderá a la petición con un código de	Sí - De lo contrario, la aplicación responderá a la petición con un código	No - Aunque la protección esté activa, no es necesario hacer uso del token dentro

	estado 419, indicando que la autenticación previamente válida del usuario ha expirado	estado 403, indicando que la aplicación considera que el usuario no tiene los permisos suficientes para hacer esa petición	de estado 403, indicando que la aplicación considera que el usuario no tiene los permisos suficientes para hacer esa petición	del formulario en las vistas para que la aplicación funcione.
GH5	Para vulnerar la aplicación contra este tipo de ataques solo hay que desactivar el middleware que viene activo por defecto	Para vulnerar la aplicación contra este tipo de ataques solo hay que desactivar el middleware que viene activo por defecto	Express no tiene protección CSRF por defecto. Se requieren librerías externas para hacer uso de esa protección	Spring se puede vulnerar si no tiene las dependencias de seguridad y/o configuraciones necesarias de la librería, en caso de que las tenga, se puede desactivar la protección agregando *.csrf().disable()* en la clase donde agregó la configuración de esa librería
GH6	https://laravel.com/docs/9.x/csrf#preventing-csrf-requests	https://docs.djangoproject.com/en/4.1/howto/csrf/	https://www.npmjs.com/package/csurf	https://www.baeldung.com/csrf-thymeleaf-with-spring-security. https://docs.spring.io/spring-security/reference/reactive/exploits/csrf.html#webflux-csrf-using

6. Discusión sobre los resultados

En la sección anterior, aplicamos las técnicas de hacking en los WAFs seleccionados para estudiar cómo se pueden proteger ante esas técnicas, a su vez que se analizó cómo se podían vulnerar para que el ataque tuviera éxito.

Pudimos observar que Laravel y Django vienen mejor preparados en cuanto a la mitigación de esas técnicas, al menos, para el tipo de aplicación que usamos para el estudio, debido a que no necesitaron componentes externos o adicionales para protegerlos de las técnicas de hacking. Para las tres técnicas seleccionadas, solo fue necesario acudir a la documentación de estos WAFs y revisar qué mecanismos podrían adoptar para reforzar la seguridad.

Podemos añadir que su estructura opinionada, al traer varios componentes, facilita al desarrollador el entendimiento de los mecanismos integrados. Sin embargo, con Spring es un caso distinto, porque, aunque su estructura sea opinionada como Laravel y Django, este no posee esos mecanismos incorporados, en su lugar, requiere que estos sean instalados o agregados como dependencias (Spring provee varias dependencias en su página oficial para su uso).

En el caso del WAF Express, al ser un WAF no opinionado y minimalista, fue necesario el uso de librerías externas, ya que la documentación del WAF, no especifica si trae incorporados algunos mecanismos de protección. Esto incorpora unos cuantos inconvenientes, ya que es al desarrollador al que manualmente le toca instalar y configurar esas librerías, y además, la librería de protección puede evolucionar a un ritmo distinto de la evolución del WAF, lo que puede generar problemas de compatibilidad. En Express, también se observó que la herramienta utilizada en ese WAF para las operaciones de bases de datos puede ser más vulnerable que las de los otros tres WAFs analizados. Ya que, en este caso, existen unos métodos y/o configuraciones que comprometan la seguridad de la aplicación. Mientras que en los otros WAFs, hay que utilizar clases o configuraciones más complejas, muy diferentes a las que vienen por defecto, lo que limita la posibilidad de vulnerabilidad.

7. Trabajos relacionados

Hay varios trabajos relacionados con la seguridad en las aplicaciones web, algunos se enfocan de manera general y otros de manera específica.

Alvarez *et al.* [16] presentan un análisis de las técnicas de hacking XSS, SQL Injection y CSRF en algunas páginas web en el ámbito colombiano. Usando el escáner automático de vulnerabilidad Acunetix, detectaron que la falla de seguridad más persistente fue CSRF y que en el sector de la educación, es donde se reportan más fallas referentes a esas 3 técnicas. Nosotros también hicimos el análisis sobre esas 3 técnicas, pero nuestro enfoque objetivo fueron los WAFs de los lenguajes más populares según GitHub 2021, a diferencia de este estudio, que su enfoque fueron los sectores y las páginas web en el país de Colombia.

Antunes *et al.* [24] presentan aproximaciones sobre la defensa en las aplicaciones web de las amenazas más comunes que son XSS y SQL Injection. Ellos indicaron que las aplicaciones web requieren tres líneas de defensa: validación de la información de entrada, de salida y protección del punto de acceso. También hicieron dos aproximaciones en detectar vulnerabilidades (análisis de caja blanca y pruebas de caja negra), por último, se mencionó la detección de ataques con sus aproximaciones y sus limitaciones.

Nithya *et al.* [25] presentan una encuesta sobre la detección y la prevención de los ataques XSS (Cross-Site Scripting). Se discutió sobre cómo los ataques XSS pueden afectar a los actuales sistemas web y, de los enfoques existentes para la prevención y detección de ese tipo de ataques, así como sus limitaciones, despliegues y aplicabilidad. La detección del ataque XSS se dividió en análisis estático, dinámico e híbrido (estático y dinámico). Y la prevención de esta técnica se dividió en tres partes: del lado del servidor, del lado del usuario y enfoques híbridos de mitigación.

Cuevas *et al.* [26] presentan un análisis en los sistemas web en los ambientes de desarrollo y producción en la Universidad Tecnológica Nacional – Facultad Regional Córdoba (UTN - FRC), se analizaron los ataques (según OWASP 2017) más utilizados, para explotar vulnerabilidades en cada etapa del ciclo de desarrollo de los sistemas (diseño, desarrollo y despliegue). Se concluyó que los mecanismos de desarrollo de autenticación fueron los que más fallaron y que hubo alto pico de *bugs* de inyecciones.

Aborujilah *et al.* [27] presentan una comparación de las técnicas de hacking XSS, SQL Injection, CSRF y Broken Authentication en los frameworks de desarrollo web backend más populares, que son: Laravel (PHP), Spring Boot (Java), Django (Python), Ruby on Rails (Ruby), y ASP.NET Core (C#). Concluyen que todos los frameworks estudiados tienen características para la protección de esas técnicas en general, a excepción de Ruby on Rails que debe usar una librería llamada Devise para Broken Authentication.

A diferencia de los trabajos citados anteriormente, nuestro estudio se enfocó en analizar tres técnicas de hacking, que fueron XSS (Cross-Site Scripting), SQL

Injection y CSRF (Cross Site Request Forgery), en los WAFs de los lenguajes de programación más populares del 2021. De los cuales, redujimos la selección a los seis lenguajes más populares y, con base en esos lenguajes, redujimos la selección de WAFs a los cuatro más populares, los cuales fueron Laravel, Django, Express y Spring. Otra diferencia fue que en nuestro estudio se implementó una aplicación web que sirvió como proyecto base, para estudiar cómo desarrollar aplicaciones tanto con protección como sin protección ante las tres técnicas de hacking anteriores. Por lo tanto, desarrollamos ocho aplicaciones web en total (cuatro con protección en cada WAF, y cuatro sin protección), y esto nos permitió realizar un estudio teórico/practico comparativo de la protección ante estas técnicas de hacking (mientras que muchos de los estudios anteriores, realizaban un estudio comparativo teórico).

8. Conclusiones

Analizamos cómo se comportan los WAFs Laravel, Django, Spring y Express contra las técnicas de hacking XSS (Cross-Site Scripting), SQL Injection y CSRF (Cross Site Request Forgery).

Para comparar el soporte que tienen esos WAFs, desarrollamos la aplicación Blog Posts en los cuatro WAFs mencionados anteriormente en dos versiones, una con protección y otra sin protección en cuanto a las tres técnicas de hacking seleccionadas.

En la versión con protección, encontramos que los WAFs Laravel y Django, proveen soporte interno contra todas las técnicas de hacking mencionadas anteriormente. Para XSS, estos WAFs tienen motores de plantilla incorporados, en ellos se usan sentencias específicas en la capa visual para la protección contra esta técnica. Para SQL Injection, estos WAFs cuentan con su propio ORM, que realizan operaciones en la base de datos y protegen automáticamente contra esta técnica. Para CSRF, estos WAFs vienen con un middleware que actúa como verificador de identidad, que protege contra esta técnica.

Por otro lado, evidenciamos que los WAFs Express y Spring, requerían el uso o instalación de librerías externas para proteger contra las técnicas de hacking mencionadas anteriormente. Para XSS, se emplearon motores de plantilla externos para proteger contra este tipo de ataques. Para SQL Injection, se emplearon ORMs externos para proteger automáticamente de este tipo de ataques. Para CSRF, se usaron librerías externas especializadas para proteger contra este tipo de ataques.

En la versión sin protección, evaluamos cómo se pueden vulnerar los WAFs ante las técnicas de hacking mencionadas anteriormente.

Para XSS, los motores de plantilla utilizados para proteger contra esta técnica pueden vulnerar la aplicación, si se usan sentencias que no permitan el escape de la información desplegada en la capa visual. Para SQL Injection, el ORM usado en el WAF Express puede ser vulnerable si se configura la conexión a la base de datos, y además se usan métodos que permitan realizar operaciones en la base de datos sin procesar. En los otros WAFs, se usaron métodos y librerías diferentes al ORM y también se configuró la conexión a la base de datos, a excepción del WAF Laravel, que en este no se configuró esa conexión. Para CSRF, en Laravel y Django, solo hubo que desactivar la protección que traen activada en el middleware por defecto. Spring y Express no vienen con esa protección activada por defecto.

Tras este estudio, concluimos que, a pesar de que actualmente hay una amplia variedad de herramientas y mecanismos para reforzar la seguridad en el mundo del desarrollo, los errores y los descuidos estarán presentes, esto conlleva a que los desarrolladores, muchas veces hagan caso omiso en cuanto a seguridad se refiere, ya sea al escribir código o usar métodos vulnerables o no recomendados, que puedan llevar algún incidente de seguridad. Por eso, nosotros recomendamos analizar con más profundidad las herramientas a usar a la hora de construir un aplicativo, optar por la mejor opción para reforzar la seguridad siguiendo las buenas prácticas, para reducir este tipo de incidentes a futuro.

Referencias

1. Okanovic, V. (2014). Web application development with component frameworks. In 2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pp. 889-892. IEEE.
2. Saleem, S. A. (2006). Ethical hacking as a risk management technique. In Proceedings of the 3rd annual conference on Information security curriculum development, pp. 201-203.
3. Szajnfarber, Z., Gralla, E. (2017). Qualitative methods for engineering systems: Why we need them and how to use them. Systems Engineering 20, pp. 497–511.
4. Velasco-Elizondo, P., Castañeda-Calvillo, L., García-Fernandez, A., & Vazquez-Reyes, S. (2017). Towards detecting mvc architectural smells, in: International Conference on Software Process Improvement. Springer, pp. 251–260.
5. GitHub Octoverse, https://octoverse.github.com/2022/top-programming-languages. Fecha de último acceso: 05/12/2022.
6. Laravel, https://laravel.com/. Fecha de último acceso: 10/09/2022.
7. Django, https://www.djangoproject.com/. Fecha de último acceso: 10/09/2022.
8. Spring, https://spring.io/. Fecha de último acceso: 10/09/2022.
9. Express, https://expressjs.com/. Fecha de último acceso: 10/09/2022.
10. Portela dos Santos, A., & Loureiro, M. (2021). Deno Web Development: Write, test, maintain, and deploy JavaScript and TypeScript web applications using Deno. Packt.
11. Khalid, M. N., Farooq, H., Iqbal, M., Alam, M. T., & Rasheed, K. (2018). Predicting web vulnerabilities in web applications based on machine learning. In International

Conference on Intelligent Technologies and Applications, pp. 473-484. Springer, Singapore.
12. Rutledge Brian. (2019). A high-level diagram of cross-site scripting. Security Boulevard, Florida, EE.UU. https://securityboulevard.com/2019/05/cross-site-scripting-xss-web-based-application-security-part-3/.
13. OWASP Foundation, XSS, OWASP, https://owasp.org/www-community/attacks/xss/. Fecha de último acceso: 01/10/2022.
14. OWASP Foundation, DOM Based XSS, https://owasp.org/www-community/attacks/DOM_Based_XSS. Fecha de último acceso: 01/10/2022.
15. OWASP Foundation, SQL Injection, https://owasp.org/www-community/attacks/SQL_Injection. Fecha de último acceso: 01/10/2022.
16. Alvarez, D. E., Correa, D. B., & Arango, F. I. (2016). An analysis of XSS, CSRF and SQL injection in colombian software and web site development. In 2016 8th Euro American conference on telematics and information systems, pp. 1-5. IEEE.
17. Oza Shyam. (2019). A high-level diagram of a SQL Injection Attack. BUSINESS 2 COMMUNITY, Londres, Reino Unido. https://www.business2community.com/cybersecurity/sql-injection-attacks-sqli-web-based-application-security-part-4-02223254.
18. OWASP Foundation, Blind SQL Injection, https://owasp.org/www-community/attacks/Blind_SQL_Injection. Fecha de último acceso: 02/10/2022.
19. OWASP Foundation, Testing for ORM Injection, https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05.7-Testing_for_ORM_Injection. Fecha de último acceso: 07/10/2022.
20. OWASP Foundation, Double Encoding, https://owasp.org/www-community/Double_Encoding. Fecha de último acceso: 07/10/2022.
21. OWASP Foundation, Code Injection, https://owasp.org/www-community/attacks/Code_Injection. Fecha último acceso: 07/10/2022.
22. OWASP Foundation, CSRF, https://owasp.org/www-community/attacks/csrf. Fecha de último acceso 01/10/2022.
23. Rutledge Brian. (2019). A diagram depicting an example of cross-site request forgery. SPANNING, Miami, Florida, EE.UU. https://spanning.com/blog/cross-site-forgery-web-based-application-security-part-2/.
24. Antunes, N., & Vieira, M. (2012). Defending against web application vulnerabilities. Computer, vol. 45(02), pp. 66-72.
25. Nithya, V., Pandian, S. L., & Malarvizhi, C. (2015). A survey on detection and prevention of cross-site scripting attack. International Journal of Security and Its Applications, vol. 9(3), pp. 139-152.
26. Cuevas, J. C., Muñoz, R. M., Gionantonio, D., Alejandra, M., Gastañaga, I., Gibellini, F., Parisi, G., Barrionuevo, D. & Zea Cárdenas, M. (2018). In XX Workshop de Investigadores en Ciencias de la Computación (WICC 2018, Universidad Nacional del Nordeste).
27. Aborujilah, A., Adamu, J., Shariff, S. M. & Long, Z. A. (2022). Descriptive Analysis of Built-in Security Features in Web Development Frameworks. In 2022 16th International Conference on Ubiquitous Information Management and Communication (IMCOM), pp. 1-8. IEEE.