Universitat Oberta de Catalunya

uoc.edu

DevSecOps: Integración de la Seguridad en Entornos CI/CD

Juan Jesús Padrón Hernández
Máster de Ciberseguridad y Privacidad Seguridad Empresarial

Miguél Ángel Flores Terron
Víctor García Font

1 de junio de 2021

FICHA DEL TRABAJO FINAL

Título del trabajo:	*DevSecOps: Integración de la seguridad en entornos CI/CD)*
Nombre del consultor/a:	*Miguel Flores Terron*
Nombre del PRA:	*Víctor García Font*
Fecha de entrega (mm/aaaa)	*06/2021*
Titulación:	*Máster en Ciberseguridad y Privacidad*
Área del Trabajo Final:	*Seguridad empresarial*
Idioma del trabajo:	*Español*
Palabras clave:	*Seguridad, DevSecOps, CI/CD*

Resumen:

La cultura DevOps y las metodologías de desarrollo ágiles han mejorado el flujo de desarrollo del software, permitiendo la implementación de nuevas funcionalidades en menor tiempo, un aspecto fundamental para destacar en el mercado frente al resto de aplicaciones. Sin embargo, muchas veces se ha sacrificado la seguridad del software desarrollado, por considerarla un freno para la implementación de nuevas características. La necesidad de contemplar la seguridad en la cultura DevOps, manteniendo la agilidad del ciclo de desarrollo ha dado lugar a DevSecOps. El objetivo de este trabajo es automatizar la seguridad en el ciclo de desarrollo del software siguiendo la cultura DevSecOps. Para esto, se ha llevado a cabo un estudio de los servidores de integración continua y de las herramientas de automatización de seguridad existentes. Se ha utilizado GitHub Actions para desarrollar una pipeline de integración continua en la que se han incluido diferentes pruebas de seguridad, entre ellas el escaneo de secretos, análisis de composición del código (SCA), pruebas de seguridad estáticas (SAST), pruebas de seguridad dinámicas (DAST) y pruebas de seguridad de la infraestructura. A pesar de las limitaciones que presenta este trabajo, relativas a la ausencia de presupuesto y a no poder probar el producto en un entorno real, los resultados son útiles para entender la cultura DevSecOps y su aplicación en el entorno empresarial. Investigaciones futuras deberían optimizar el rendimiento del producto obtenido además de incluir herramientas de gestión de logs y monitorización, con el fin de controlar el funcionamiento del software en producción.

Abstract:

The DevOps culture and agile development methodologies have improved the software development flow, allowing the implementation of new functionalities in less time, a fundamental aspect to stand out in the market compared to other applications. However, the security of the developed software has often been sacrificed, because it has been considered a brake for the implementation of new features. The need to consider security in the DevOps culture while maintaining the agility of the development cycle has given rise to DevSecOps. The objective of this work is to automate security in the software development cycle following the DevSecOps culture. For this purpose, the project studies the existing continuous integration servers and security automation tools. GitHub Actions has been used to develop a continuous integration pipeline in which different security tests have been considered, including secrets scanning, Software Composition Analysis (SCA), Static Application Security Testing (SAST), Dynamic Application Security Testing (DAST), and infrastructure security testing. Despite the limitations of this work, related to the lack of budget and not being able to test the product in a real environment, the results are useful for understanding the DevSecOps culture and its application in the enterprise environment. Future research should optimize the performance of the product obtained and include log management and monitoring tools, in order to control the operation of the software in production.

Índice

Índice de figuras

Índice de cuadros

1. Introducción

1.1. Contexto y justificación del trabajo

La ciberseguridad ha ido cobrando cada vez más importancia durante los últimos años. La actual crisis sanitaria mundial debido a la COVID-19, y las medidas tomadas para combatir la situación, como el confinamiento domiciliario, el distanciamiento social o el cierre de comercios, han hecho que la tecnología juegue un papel primordial para la sociedad. Esta situación extraordinaria ha supuesto una disrupción en la vida diaria de las personas y en la manera de relacionarnos. Una de las áreas que se ha visto más afectada es la laboral. Ante la necesidad de fomentar el teletrabajo y de digitalizar los modelos de negocio, muchas empresas han adoptado las tecnologías de la información de una manera apresurada, muchas veces descuidando aspectos esenciales como la seguridad informática, lo que ha supuesto, según datos del FBI [1], un incremento de los ciberataques en un 400 % durante la pandemia.

Asimismo, las metodologías de desarrollo de software ágiles se han vuelto el estándar, cubriendo las necesidades de los clientes en un mercado dinámico en el que cada vez nacen más tecnologías y funcionalidades que pueden marcar la diferencia con la competencia. A esto se ha sumado el auge de la computación en la nube y de la arquitectura de microservicios, la cual estructura una aplicación como un conjunto de pequeños servicios desacoplados, desplegables de forma independiente y desarrollados por pequeños equipos [2]. De esta manera ha aumentado la complejidad a nivel de infraestructura de cada producto, teniendo que gestionarse tanto la configuración como el despliegue y la comunicación de estos.

En este contexto surge la cultura DevOps [3], que trata de aunar los esfuerzos entre los equipos de desarrollo y de operaciones, con el fin de automatizar procesos como pueden ser el testeo, el despliegue o la monitorización, haciéndolos presentes en todo el ciclo de desarrollo del software. La agilidad en el ciclo de desarrollo del software permite que se realice un mayor número de entregas y despliegues [4]. Sin embargo, en muchos casos se descuida la seguridad porque se considera un freno para el ciclo del desarrollo del software [5] y menos de un 20 % de compañías que siguen la cultura DevOps la tienen en cuenta [5]. Lo cierto es que los equipos de desarrollo suelen tardar de media más tiempo en verificar y solucionar las vulnerabilidades encontradas en el software en producción que cuando este se encuentra en desarrollo [4]. Por tanto, es importante tener presente la seguridad desde las etapas más tempranas del ciclo de vida del producto, evitando asimismo posibles brechas de seguridad y las nefastas implicaciones de estas.

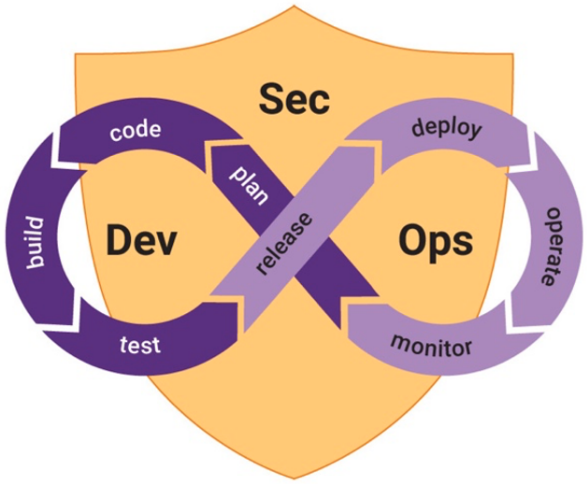

Figura 1.1: Ciclo de desarrollo del software DevSecOps

Es por la necesidad de tener en cuenta la seguridad en DevOps que surge DevSecOps, y pasa a ser uno de los pilares de esta culturas promoviendo las buenas prácticas y la colaboración entre desarrolladores, administradores de sistemas y expertos en ciberseguridad en todo el ciclo de desarrollo [6]. Para conseguir esto de una forma eficiente es imprescindible la automatización de los procesos relacionados con la seguridad, como puede ser el *hardening* de la infraestructura sobre la que se realizarán los despliegues, el análisis tanto del código y sus dependencias como de la infraestructura, y el registro de métricas y alertas para situaciones críticas.

Este trabajo parte de la necesidad que tienen las empresas emergentes por tener en consideración la seguridad durante su ciclo de desarrollo del software, por lo que es importante conocer las tecnologías que más emplean para el desarrollo de sus sitios webs y aplicaciones. Atendiendo al reporte anual The State of Developer Ecosystem llevado a cabo por JetBrains en el año 2020 [7], JavaScript es el lenguaje más usado, pues un 70 % de los desarrolladores encuestados votaron haberlo usado en algún proyecto en el último año y un 39 % lo definió como su lenguaje principal. Si bien es cierto que su uso mayoritario se debe a su actual dominio en tecnologías frontend, este mismo reporte muestra que la mayoría de desarrolladores, un 69 %, trabajan de lado del backend. Es por esto que se ha optado por trabajar sobre una aplicación backend desarrollada con JavaScript sobre el entorno de ejecución Node.js [8].

1.2. Objetivos del trabajo

El objetivo general de este trabajo es desarrollar una pipeline que sea de utilidad a las empresas para tener presente la seguridad en todo el ciclo de desarrollo del software, automatizando los procesos implicados en esta, con el fin de detectar los problemas de seguridad desde las etapas más tempranas del desarrollo. Se engloban dentro de este propósito general los siguientes objetivos secundarios, que se dividirán en objetivos de investigación y objetivos de desarrollo.

Objetivos de investigación:

1. Estudiar el estado actual de DevOps y DevSecOps comprendiendo las bases de estos movimientos y analizando las herramientas más usadas.

2. Estudiar los aspectos de seguridad a tener en cuenta en aplicaciones Node.js y cómo detectarlos con herramientas automáticas.

3. Analizar las herramientas disponibles en el mercado para implementar integración y entrega continua, valorando las ventajas de cada una sobre su competencia con el fin de elegir la más adecuada para el producto final.

Objetivos de desarrollo:

1. Configurar la herramienta de integración continua elegida para automatizar los procesos de construcción y de las pruebas de seguridad posteriores.

2. Integrar las herramientas de pruebas de seguridad, tanto de estáticas como dinámicas y de infraestructura.

3. Automatizar el despliegue de la infraestructura de forma segura empleando herramientas de infraestructura como código (IaC).

1.3. Enfoque y método seguido

Como este proyecto sigue una perspectiva empresarial, se desarrollará en primer lugar una aplicación para tomar como producto de partida. En segundo lugar, se estudiarán en profundidad las tecnologías a emplear en este proyecto. Esto servirá para planificar la fase de implementación y para diseñar el producto final. En tercer lugar, se desarrollarán y configurarán los elementos necesarios para integrar las tecnologías en la pipeline de CI/CD, y se probará su funcionamiento. Por último, se elaborarán las conclusiones en base a los resultados obtenidos y a la información recopilada, determinando la viabilidad de la solución desarrollada en un entorno empresarial.

1.4. Planificación del trabajo

La planificación del trabajo aparece reflejada en el Cuadro 1.1, así como en el Diagrama de Gantt de la Figura 1.2.

Cuadro 1.1: Planificación temporal

Actividad	Inicio	Fin	Días
Entrega 1 – Plan de trabajo	**17/02/2021**	**02/03/2021**	**14**
Contexto y justificación del trabajo	17/02/2021	21/02/2021	5
Definición de objetivos y enfoque	22/02/2021	25/02/2021	4
Planificación del trabajo	26/02/2021	26/02/2021	1
Revisión del estado del arte	27/02/2021	02/02/2021	4
Entrega 2 – Investigación y diseño	**03/03/2021**	**30/03/2021**	**28**
Investigación DevOps	03/03/2021	04/03/2021	2
Investigación DevSecOps	05/03/2021	07/03/2021	3
Investigación integración continua	08/03/2021	10/03/2021	3
Elección servidor CI/CD	11/03/2021	12/03/2021	2
Investigación herramientas automatización de la seguridad	13/03/2021	17/03/2021	5
Elección y documentación de herramientas de automatización de la seguridad	18/03/2021	22/03/2021	5
Investigación y elección de herramientas de escaneo de infraestructura	23/03/2021	27/03/2021	5
Diseño de la pipeline CI/CD	28/03/2021	30/03/2021	3
Entrega 3 – Implementación	**31/03/2021**	**27/04/2021**	**28**
Configuración de la infraestructura y el despliegue	31/03/2021	03/04/2021	4
Configuración pipeline en servidor CI/CD	04/03/2021	07/04/2021	4
Integración herramientas escaneo de secretos	08/04/2021	11/04/2021	4
Integración herramientas SAST	12/04/2021	15/04/2021	4
Integración herramientas DAST	16/04/2021	19/04/2021	4
Integración herramientas escaneo de infraestructura	20/04/2021	23/04/2021	4
Configuración despliegue continuo	24/04/2021	27/04/2021	4
Entrega 4 – Presentación final	**28/04/2021**	**18/06/2021**	**52**
Análisis del producto final obtenido	28/04/2021	04/05/2021	7
Elaboración de conclusiones del TFM	05/04/2021	11/05/2021	7
Completar memoria final	12/05/2021	01/06/2021	21
Presentación en vídeo	02/06/2021	08/06/2021	7
Preparación de la defensa	09/06/2021	13/06/2021	5
Defensa del TFM	14/06/2021	18/06/2021	5

Figura 1.2: Ciclo de desarrollo del software DevSecOps

1.5. Estado del arte

Herramientas de DevOps

Desde que en 2009 surge la idea de DevOps en la conferencia "10+ Deploys per Day: Dev and Ops Cooperation at Flickr" [10], se ha desarrollado una gran cantidad de herramientas cuyo objetivo es la automatización de procesos durante el ciclo de desarrollo del software. Por un lado, destacan las herramientas de integración, de entrega y de despliegue continuos (CI/CD) [13], que permiten la automatización de diferentes procesos durante el desarrollo del software. La integración continua, según Martin Fowler, "es una práctica del desarrollo de software donde los miembros de un equipo integran su trabajo frecuentemente, y cada integración es verificada mediante un proceso automatizado de construcción y testeo" [11]. La entrega continua se consigue gracias a la integración continua, y consiste en que el código siempre esté preparado para desplegarse en producción. El despliegue continuo, por su parte, es el proceso por el cual un cambio en el código se despliega automáticamente tras pasar por las pruebas automatizadas [12]. Las herramientas de automatización CI/CD son los servidores de integración continua. Existe una gran variedad de estos servidores en el mercado, entre ellos se encuentran GitLab CI, Circle CI, Travis, o el más reciente, GitHub Actions. Según el DevOps Report del años 2020 llevado a cabo por GitKraken [14], Jenkins es el servidor de integración continua más usado. No obstante, GitHub Actions es una apuesta fuerte a pesar de ser la herramienta más reciente, ya que facilita la automatización gracias a la denominadas acciones [9] y a un *marketplace* donde la comunidad publica las automatizaciones realizadas para que otras personas puedan usarlas en sus *pipelines*.

Por otro lado, son destacables las herramientas de gestión de infraestructura, como Ansible y Terraform, herramientas de gestión de configuraciones y aprovisionamiento de recursos de la infraestructura. Asimismo, entre las herramientas de gestión de infraestructura, los contenedores se han convertido en el estándar en lo referente a entornos de ejecución del software. El gestor de contenedores más conocido es, sin lugar a dudas, Docker, una herramienta de código abierto desarrollada por Docker Inc. y lanzada en el año 2013 [15], aunque existen otras alternativas menos populares como Podman, desarrollado por RedHat [16] y cuyo uso es similar. Gracias a los contenedores, es posible disponer de entornos similares en desarrollo y producción, asegurando que el funcionamiento del software a desarrollar sea prácticamente igual en ambos. Para poder desplegar de una forma eficiente y robusta los contenedores en producción surgieron los orquestadores de contenedores, como Docker Swarm y Kubernetes, siendo este último el más popular. Kubernetes fue desarrollado por Google y publicado en el año 2014, y facilita la configuración de los contenedores y el entorno donde se ejecutan así

como la automatización del despliegue de los mismos.[17].

Herramientas de DevSecOps

La OWASP DevSecOps Guideline [18] desataca cinco tipos de herramientas de automatización de la seguridad a aplicar en diferentes partes del ciclo de desarrollo del software, descritas en el siguiente capítulo de la memoria 2.2. Algunos ejemplos concretos de herramientas son OWASP Zed Attack Proxy, Semgrep, TruffleHug y Anchore.

2. Marco Teórico

En este capítulo se desarrollarán los aspectos teóricos así como la investigación realizada y el diseño del producto que se implementará en el siguiente capítulo.

2.1. DevOps

Existen muchas definiciones diferentes de DevOps disponibles en libros, en artículos de revistas o en Internet. A raíz de esta disparidad en las definiciones, surgen diversos estudios que tratan de darle una descripción académica [22] [23]. Según estos estudios, podemos definir DevOps como una cultura que trata de aunar a los equipos de desarrollo y operaciones, basándose en una serie de principios y prácticas [23] que pretenden acelerar las entregas del producto mejorando el *feedback* de los clientes y la capacidad de reacción ante los cambios [21].

El término DevOps surge a finales de los 2000, en un contexto en el que las metodologías de desarrollo ágiles cada vez tomaban mayor relevancia en la industria del desarrollo de software. La velocidad a la que se desarrollaban nuevas características o se corregían *bugs* distaba mucho de la velocidad a la que se realizaban los despliegues de estos cambios, con lo que el ciclo de desarrollo del software se veía ralentizado. Esta ralentización era resultado de la falta de comunicación existente entre el equipo de desarrollo y el de operaciones. Fue Patrick Debois quien, en 2007, tras una experiencia frustrante trabajando en la migración un gran centro de datos, se percató de cómo esta falta de comunicación entre desarrolladores y administradores de sistemas afectaba al flujo de trabajo [24]. En 2009, John Allspaw y Paul Hammond, ingenieros de Flickr, presentaron su charla *"10 Deploys a Day: Dev and Ops Cooperation at Flickr"* [10], donde propusieron integrar desarrollo y operaciones en un flujo automatizado. Patrick Debois, tras esta conferencia, decidió organizar una similar en Bélgica, a la que llamó DevOpsDays, de donde surge el término DevOps [25].

Un aspecto importante para seguir exitosamente la cultura DevOps es la automatización de procesos, al ser lo que permite mantener la agilidad durante el desarrollo del software. La importancia de la automatización de procesos ha hecho surgir una gran cantidad de herramientas que contemplan la construcción del software, la integración y el despliegue continuos, la gestión de *logs* y la monitorización [26].

2.1.1. Herramientas de Construcción

Las herramientas de construcción abarcan diferentes puntos del ciclo de vida del desarrollo del software: la compilación o transpilación, el manejo de dependencias

y la ejecución de tests[3], entre otros. Estas herramientas suelen estar orientadas a un lenguaje específico, como por ejemplo Rake para Ruby, Gradle o Maven para Java, Cargo para Rust o NPM para Javascript y Typescript.

2.1.2. Despliegue y Configuración

Las herramientas de despliegue y configuración tienen como fin automatizar la configuración de la infraestructura sobre la que se ejecutará la aplicación, así como los despliegues sobre la misma. La gran ventaja de estas herramientas es la de mantener siempre una misma configuración, minimizando las diferencias entre entornos de desarrollo, preproducción y producción [26]. Estas herramientas plantean la idea de tratar la infraestructura como código [3], surgiendo así el concepto de IaaS, por sus siglas en inglés *Infrastructure as a Service*. Aquí destacan herramientas como Ansible, Puppet, Terraform, o herramientas de gestión y orquestación de contenedores como Docker y Kubernetes.

2.1.3. Gestión de Logs

Los *logs* son utilizados para analizar el rendimiento del sistema, para detectar patrones de uso en la aplicación, y para encontrar errores, entre otros usos. Son críticos para medir el éxito del servicio y de las actualizaciones que se van aplicando. Los *logs* tomados en las primeras etapas del ciclo de vida del software sirven tanto para encontrar errores en el código como para analizar el rendimiento de éste, mientras que si se toman en etapas más avanzadas de este ciclo, pueden servir para monitorizar la aplicación, para encontrar problemas en el entorno de producción y para tomar datos analíticos sobre el uso de la aplicación [26]. Algunos ejemplos de estas herramientas son Logstash, Fluentd, Loggly o Graylog.

2.1.4. Monitorización

Las herramientas de monitorización se emplean para determinar si un software desplegado funciona de forma óptima. Pueden clasificarse en herramientas de monitorización de sistemas y en herramientas de monitorización de red. [26]. Las herramientas de monitorización de sistemas nos ayudan a conocer aspectos como el uso CPU, RAM, o espacio de disco en el sistema, mientras que la monitorización de red se centra en el tráfico de la misma [3]. Entre estas podemos destacar Nagios, New Relic o Prometheus.

2.2. DevSecOps

El crecimiento de las metodologías ágiles de desarrollo del software y la acogida de la cultura DevOps por parte de las organizaciones, ha incrementado la velocidad a la que las aplicaciones reciben actualizaciones. Esto tiene grandes ventajas, pues permite tener *feedback* temprano del cliente para mejorar el producto desde las primeras fases del desarrollo, mejorando la adaptabilidad del producto con el entorno y permitiendo marcar la diferencia con la competencia gracias a la implementación de nuevas funcionalidades o a la mejora del funcionamiento de las ya existentes [27]. Sin embargo, a veces esta agilidad se consigue a costa de sacrificar otros aspectos del producto final, como puede ser la seguridad [28]. Los estudios muestran que menos de un 20 % de las compañías que siguen la cultura DevOps tienen en cuenta la seguridad como parte del ciclo de desarrollo del software [5].

DevSecOps surge para incluir la seguridad en DevOps, alineando los equipos de desarrollo, de operaciones y de seguridad durante todo el ciclo de desarrollo. Esto se consigue desplazando la seguridad a la izquierda, es decir, considerándola desde las primeras etapas del desarrollo [28]. De esta manera, al tenerla en cuenta desde el diseño de la aplicación, es posible realizar los controles de seguridad necesarios a lo largo del ciclo de desarrollo del software y automatizarlos para que sean rápidos, escalables y efectivos [29]. De esta manera se mantiene la agilidad en el desarrollo del software y se detectan desde fases tempranas los fallos de seguridad que, de llegar al cliente, conllevarían grandes perdidas de tiempo y dinero.

Al igual que en DevOps, las herramientas juegan un papel de gran importancia. Existen una gran cantidad de herramientas para llevar a cabo DevSecOps. Se ha tomado como referencia la guía *OWASP DevSecOps Guideline* [18] para establecer los aspectos más importantes relativos a la seguridad: la detección de secretos, las Pruebas Estáticas de la Seguridad de la Aplicación (SAST), las Pruebas Dinámicas de la Seguridad de la Aplicación (DAST), el escaneo de la infraestructura y la comprobación del cumplimiento normativo.

2.2.1. Detección de secretos

Un error común a la hora de desarrollar código es dejar contraseñas, claves privadas, API KEYS u otros secretos explícitamente en el código, subiéndolos muchas veces al repositorio con las implicaciones para la seguridad que esto conlleva si otras personas, ya sean de la misma empresa o externas, tienen acceso a dicho repositorio. Las herramientas de detección de secretos se encargan de analizar todos los ficheros del repositorio buscando cadenas de texto que podrían contener secretos. Entre estas comprobaciones pueden contemplarse aspectos como

la búsqueda de variables nombradas como *password*, *credentials* o *API_KEY* inicializadas a una cadena de texto explícita, cadenas de conexión SSH, o cadenas que por su carácter aleatorio puedan representar contraseñas cifradas. Estas herramientas se pueden combinar con otras para automatizar su ejecución antes de subir los cambios al repositorio remoto, evitando la publicación errónea de datos sensibles [20]. Existen muchas herramientas disponibles para este propósito, algunas ofrecidas por los propios repositorios de código como en el caso de GitLab [30] o GitHub [31], y otras herramientas ajenas a éstos, como TruffleHog. Además, cabe mencionar la existencia de herramientas para la gestión segura los secretos, como git-secret, Hashicorp Vault, Confidant o AWS Secret Manager.

2.2.2. Pruebas Estáticas de la Seguridad de la Aplicación (SAST)

Las pruebas estáticas de seguridad analizan el código fuente de la aplicación sin ejecutarlo, tratando así de encontrar vulnerabilidades o *bugs* [32]. Los análisis estáticos se pueden realizar de diferentes formas, desde las más sencillas y rápidas que contemplan solo un análisis del código fuente en base a unas serie de reglas establecidas, hasta análisis más complejos que construyen el árbol abstracto de sintaxis del programa y buscan en él patrones vulnerables [32]. Los factores a tener en cuenta a la hora de decantarse por una herramienta u otra son la velocidad a la que se quiere realizar el análisis y la profundidad del mismo, pues a más profundidad, más se tardará en realizar y viceversa. Además, se ha de considerar el porcentaje de falsos positivos que puede señalar la herramienta y el lenguaje de programación de la aplicación. Las herramientas SAST ayudan a detectar vulnerabilidades como inyecciones SQL, *cross-site scripting* y problemas de gestión de memoria, entre otras. Algunos ejemplos de estas herramientas son Semgrep, CodeSonar o CodeQL.

Como parte de las herramientas SAST en este proyecto se han incluido las herramientas de Análisis de Composición del Software (Software Composition Analysis; SCA). Las herramientas SCA inspeccionan las dependencias externas que tiene el código de una aplicación en busca de vulnerabilidades, normalmente apoyándose en una gran base de datos en la que se reportan vulnerabilidades existentes [33]. Son especialmente relevantes en entornos como Node.js, donde se hace gran uso de librerías externas y, el hecho de tener una gran comunidad desarrollando diferentes paquetes que nos ahorran desarrollar toda funcionalidad desde cero, también implica una mayor superficie de amenaza donde los atacantes puede encontrar vulnerabilidades. De hecho, el propio gestor de paquetes de Node.js NPM dispone de una funcionalidad que audita las dependencias instaladas [65].

2.2.3. Pruebas Dinámicas de la Seguridad de la Aplicación (DAST)

A diferencia de las pruebas estáticas, estas pruebas de seguridad se realizan contra las aplicaciones en ejecución sin conocer su código fuente, por lo que son consideradas pruebas de caja negra [35]. Las herramientas de análisis dinámico de la seguridad lanzan peticiones a la aplicación como si fueran atacantes, tratando de vulnerarla y detectando así errores de configuración y cabeceras HTTP inseguras, entre otras vulnerabilidades. La herramienta más conocida de este tipo es OWASP ZAP, aunque existen alternativas como StackHawk.

2.2.4. Escaneo de la infraestructura

Un elemento clave a considerar al poner en producción una aplicación es la infraestructura sobra la que esta se desplegará. La configuración de la infraestructura no solo afecta al rendimiento y a la fiabilidad del servicio, sino que además juega un papel fundamental en la seguridad del mismo. Por muy seguro que sea el código de una aplicación, si ésta se encuentra desplegada en una infraestructura poco segura, los atacantes podrán vulnerarla. En los últimos años, gracias a la adopción de DevOps, la Infraestructura como Código (Infrastructure-as-Code; IaC) se ha convertido en la práctica más recomendable a la hora de configurar y automatizar la infraestructura [36]. La Infraestructura como Código aborda la configuración y la automatización de la infraestructura como si de un desarrollo de software se tratase, con las ventajas inherentes a este proceso (reusabilidad, control de versiones,...) [37]. Algunas de las herramientas IaC más conocidas son Chef, Puppet y Ansible.

Dentro de las herramientas de configuración de la infraestructura destacan las herramientas de gestión de contenedores, que permiten empaquetar aplicaciones en imágenes con el fin de compartirlas, ya sea públicamente o entre entornos en un mismo proyecto privado. Estas imágenes pueden ser vulnerabilidades, ya sea por su configuración base o por el código, por dependencias o por configuraciones adicionales concretas del proyecto [39]. Las herramientas de escaneo de imágenes tratan de encontrar estas vulnerabilidades desde su generación, evitando que se empleen siendo inseguras.

2.2.5. Comprobación del cumplimiento normativo

El último tipo de herramientas considerado por OWASP es el de la comprobación del cumplimiento normativo o *compliance check*. En la actualidad existen un gran número de estándares y normativas de seguridad, muchos de obligado cumplimiento por parte de las empresas. Se pueden realizar las comprobaciones

requeridas por estas normativas de una forma automatizada gracias a herramientas como Inspect, DevSec Hardening Framework o Serverspec [20].

2.3. Integración Continua y Despliegue Continuo

La integración continua (CI) surge como una de las prácticas de la metodología ágil Extreme Programming (XP), en la cual se propone que los desarrolladores publiquen sus cambios varias veces al día en el repositorio de código. De esta forma pueden encontrarse problemas de compatibilidad entre estos cambios en etapas más tempranas del desarrollo, y se evitan complejos y largos procesos de integración en los días anteriores de la fecha de entrega del proyecto o del hito [40]. Gracias a estas ventajas, la integración continua se utiliza como práctica independiente de XP, respaldada por referentes en el mundo del desarrollo del software como Martin Fowler [41].

El despliegue continuo (CD) amplía las bases propuestas por la integración continua, proponiendo no solo la integración automatizada del código en el repositorio, sino también el despliegue automatizado del mismo en el entorno de producción [44]. La automatización del despliegue es especialmente beneficiosa cuando existen varios entornos en los que se ha de desplegar el software cuando se genera una nueva versión, y cuando este proceso de despliegue ocupa mucho tiempo [47]. Cabe clarificar las diferencias entre la entrega continua y el despliegue continuo, conceptos que en ocasiones se confunden. Mientras la entrega continua tiene como objetivo mantener siempre el software en un estado que permite su despliegue inmediato [48], el despliegue continuo implica el despliegue de las nuevas versiones del software de forma automática.

Los procesos automatizados de integración y de despliegue pueden tardar varios minutos en ejecutarse, ralentizando el flujo de desarrollo. Además, requieren de una gran coordinación por parte de los desarrolladores para evitar conflictos en los cambios y en el orden en que se realizan las integraciones. Por estos motivos, desde la metodología XP se propone el uso de un servidor dedicado a realizar las integraciones. Este servidor, denominado servidor de integración continua, asegura la realización de los procedimientos necesarios para integrar los nuevos cambios de manera ordenada y evitando conflictos [42] y su elección es clave para llevar a cabo con éxito estos procesos. Existen varias alternativas en el mercado, siendo GitLab CI, Jenkins y GitHub Actions los servidores de integración continua más destacables. Tanto GitLab CI como GitHub Actions forman parte del ecosistema de los gestores de repositorios GitLab y GitHub respectivamente, por lo que son los que se tendrán en cuenta en este proyecto, evitando así la dependencia de una herramienta adicional para este propósito.

GitHub Actions frente a GitLab CI

Para valorar estos servidores de integración continua, se compararán a continuación atendiendo a los siguientes parámetros: madurez, funcionalidades, facilidad de uso, versatilidad y precio. A la hora de comparar estos servidores de integración continua, hemos de establecer una serie de parámetros a valorar. En este caso, se establecerán los siguientes: madurez, funcionalidades, facilidad de uso, versatilidad y precio.

En lo que respecta a la madurez de ambos proyectos, la primera versión de GitLab CI data de noviembre de 2012, mientras que GitHub Actions surge en noviembre de 2019. Con 7 años de ventaja, GitLab CI es un proyecto de gran madurez, elegido por grandes compañías como la Agencia Espacial Europea (ESA) [49].

En cuando a las funcionalidad, a pesar de que GitHub Actions sea más reciente, presenta prácticamente las mismas funcionalidades que GitLab CI. Ambos permiten definir diferentes *pipelines*, ejecutar *jobs* en paralelo, establecer dependencias entre *jobs*, el manejo de artefactos o el uso de cachés para agilizar las *pipelines*. GitHub Actions presenta alguna novedad como la flexibilidad en los eventos que pueden accionar la ejecución de la *pipeline*. Los posibles eventos van desde los más comunes, como puede ser la subida de código al repositorio (push) y la creación de una *pull request*, hasta casos más específicos, como cuando se hace un *fork* del repositorio o cuando se hace un comentario en una *issue* [50].

La facilidad de uso es una de las grandes ventajas de GitHub Actions frente a GitLab CI. A pesar de que ambas herramientas se definan mediante ficheros YAML con sintáxis parecidas, la solución propuesta por GitHub destaca por las denominadas acciones o *actions*. La documentación oficial de GitHub define a las acciones como tareas individuales que se pueden combinar para crear *jobs* [51]. Además de crear acciones propias, permite utilizar las creadas por otras personas y compartirlas en un *marketplace* público. Esto hace que las integraciones de diferentes herramientas dentro de la *pipeline* se simplifique considerablemente, pues muchas veces los propios desarrolladores de dicha herramienta desarrollan una acción oficial, o existen acciones de la comunidad.

Respecto a la versatilidad, ambas herramientas ofrecen tanto la solución de ejecutar las *pipelines* en sus propios servidores, como la de ejectutarlas *on-premise*. GitHub Actions permite la ejecución en servidores de GitHub no solo en sistemas Linux, sino también Windows y MacOS, mientras que GitLab CI, solo ofrece servidores Linux.

Por último, en cuanto al precio, ambas herramientas ofrecen tres planes de monetización: uno gratuito, uno pensado para equipos, y otro pensado para organizaciones. Estos planes no solo incluyen el uso de los servidores de integración

continua de GitHub y GitLab, sino que además se incluye el uso de estas plataformas como repositorios de código, y otra serie de funcionalidades, como gestores de tareas, *wikis*, etc. El Cuadro 2.2 refleja la comparación de cada uno de estos planes teniendo en cuenta las principales características que afectan a la solución de integración continua. Hay que tener en cuenta que los valores que figuran en el cuadro solo aplican a proyectos privados, ya que en el caso de GitHub, su plan gratuito no presenta los límites establecidos a los repositorios públicos.

Cuadro 2.2: Comparativa de planes GitHub y GitLab

	GitHub	**GitLab**
Gratis		
Precio al mes por usuario en euros	0	0
Minutos al mes CI/CD	2000	400
Almacenamiento de paquetes	500MB	10GB
Ramas protegidas	Repos. públicos	Sí
Seguridad extra	No	No
Equipos		
Precio al mes por usuario en euros	3.28	15.56
Minutos al mes CI/CD	3000	10000
Almacenamiento de paquetes	2GB	10GB
Ramas protegidas	Sí	Sí
Seguridad extra	No	No
Empresarial		
Precio al mes por usuario en euros	17	81.20
Minutos al mes CI/CD	50000	50000
Almacenamiento de paquetes	50GB	10GB
Ramas protegidas	Sí	Sí
Seguridad extra	Sí	Sí

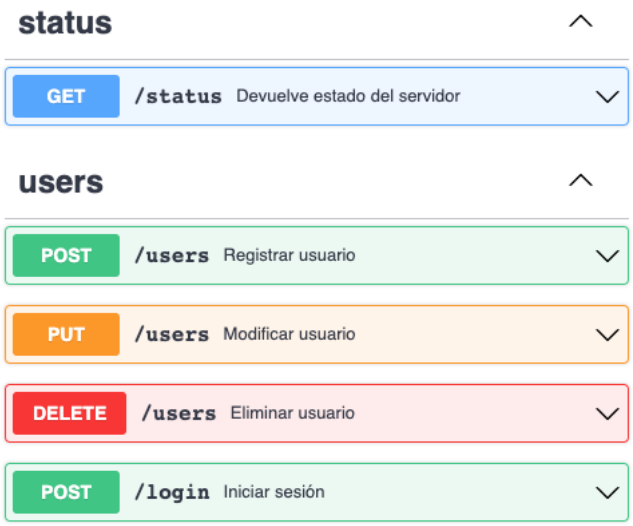

Figura 3.3: Definición de rutas del API

3. Diseño de la Solución

3.1. Supuesto empresarial

Para el desarrollo de este proyecto de ha partido del supuesto de una empresa de comercio electrónico especializada en la venta de libros. Su equipo de desarrollo ha adoptado la cultura DevOps, pero a raíz del incremento de ataques informáticos, han decidido incluir controles de seguridad en su ciclo de desarrollo. El primer servicio que han decidido asegurar es su API de autenticación, un servicio desarrollado en Node.js versión 14, que gestiona el registro, el inicio de sesión, la modificación y el borrado de usuarios. Los datos de los usuarios se almacenan en una base de datos PostgreSQL. Podemos ver las rutas definidas en la Figura 3.3.

3.2. Diseño del workflow

En la Figura 3.4 podemos ver el diseño del *workflow* de GitHub Actions planteado. Este se compone de 5 fases:

- **Fase 1:** En esta fase se ejecutan 3 *jobs* en paralelo. El primero con la herramienta seleccionada para el escaneo de secretos, el segundo con las Herramientas de pruebas estáticas de la seguridad de la aplicación, y el tercero con los *tests* del código.
- **Fase 2:** En esta fase se construye la imagen Docker de la aplicación. Para

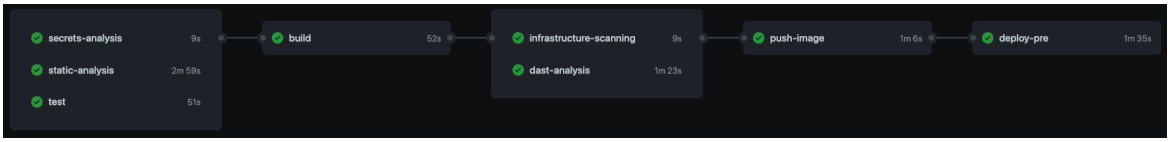

Figura 3.4: Diseño *workflow* GitHub Actions

que se ejecute esta fase tiene que haber terminado la ejecución de la fase 1.

- **Fase 3:** Una vez se haya generado la imagen, se ejecuta esta fase, en la cual se realiza el escaneo de la infraestructura y se ejecuta la herramienta de análisis dinámico de la seguridad de la aplicación.
- **Fase 4:** En este punto se sube la imagen al repositorio de imágenes.
- **Fase 5:** Por último, se realiza el despliegue de la aplicación.

3.3. Herramientas Empleadas

3.3.1. Servidor de Integración Continua

Atendiendo al resultado de la comparación de ambos servidores expuesta en el apartado 2.3, se ha escogido como servidor de integración continua GitHub Actions. Se ha considerado que los planes diseñados para equipos y para empresas de GitHub tienen un precio más reducido que los de GitLab. Partiendo del supuesto empresarial definido para este trabajo, el plan ideado para equipos de GitHub es adecuado, y supondría un ahorro de 12.28€ al mes por cada desarrollador que participase en el proyecto frente al mismo plan en GitLab. Inicialmente, las limitaciones de minutos al mes para la ejecución de los flujos de integración continua y de capacidad de almacenamiento de paquetes no suponen un problema, y en el caso de llegar a suponerlo en un futuro, se podría mejorar el plan optando por el empresarial. Además, la facilidad de configuración de GitHub Actions facilita centrar los esfuerzos en las funcionalidades del software a desarrollar. Cabe señalar que, aunque GitLab CI es el más utilizado en el sector empresarial, esto se debe principalmente a que GitLab permite su despliegue *on-premise* de forma gratuita. Sin embargo, este aspecto no se ha tenido en cuenta para este proyecto.

Configuración de GitHub Actions

La configuración de *pipelines*, conocidas en GitHub Actions como flujos de trabajo o *workflows*, se realiza en un fichero YAML situado en el directorio .github/-

workflows en la raíz del proyecto. Cada *workflow* está compuesto por su nombre -indicado en la etiqueta *name*-, el evento que los acciona -indicado en la etiqueta *on*-, y los *jobs* que lo componen -indicados en la etiqueta *jobs*-. Son varios los eventos que pueden accionar un *workflow*, y pueden encontrarse en la documentación oficial [50]. Cada *workflow* está compuesto por uno o más *jobs*. Puede definirse en qué tipo de sistema operativo se quiere ejecutar el *job* gracias a la etiqueta *runs-on*, siendo Ubuntu, Windows y MacOS los compatibles. Cada *job* se divide a su vez en diferentes pasos o *steps*. Cada *step* puede definirse por un conjunto de comandos o por una acción, ya sea propia o de un tercero. Puede encontrarse información más detallada de la sintáxis de configuración de *workflow* en la documentación oficial proporcionada por GitHub [52].

3.3.2. Herramientas DevOps

A continuación se explicarán las herramientas relacionadas con la cultura DevOps empleadas en el proyecto siguiendo la clasificación establecida en la investigación 2.1. Según esta clasificación, podemos distinguir cuatro tipos de herramientas: las de construcción, las de despliegue y configuración, las de gestión de *logs*, y las de monitorización. A pesar de ser herramientas de gran importancia en DevOps, en este proyecto no se han abordado ni las herramientas de gestión de *logs* ni las de monitorización, dando prioridad a los aspectos que afectan principalmente a la integración y despliegues continuos, es decir, a las herramientas de construcción y a las de despliegue y de configuración.

Herramientas de construcción

Dado que se el supuesto empresarial parte de una aplicación Node.js, se ha utilizado como herramienta de construcción Node Package Manager (NPM). NPM surgió como un gestor de paquetes de Node.js. Además de gestor de paquetes, tiene funcionalidades para gestionar aspectos como la ejecución del programa en diferentes entornos, la ejecución de *tests*, la generación de documentación mediante paquetes externos, o la transpilación del código cuando se emplea junto a Typescript.

Herramientas de despliegue y configuración

Por un lado, como herramienta de configuración se ha empleado la herramienta de gestión de contenedores Docker. Esta herramienta permite empaquetar aplicaciones junto a las librerías y las dependencias necesarias, para posteriormente ejecutarlas como un paquete único denominado contenedor [38]. Gracias

a esto, se puede ejecutar la aplicación en cualquier entorno compatible con la herramienta de contenedores escogida. El paquete generado se denomina imagen. Cada imagen está compuesta por diferentes capas que parten de una imagen base sobre la que se van añadiendo diferentes configuraciones. Las herramientas de gestión de contenedores se han convertido en un estándar en la industria tanto en entornos de desarrollo como en entornos de producción, gracias a las herramientas de orquestación de contenedores como Kubernetes. Este éxito se debe a la capacidad que ofrecen para replicar entornos de una forma eficiente y rápida.

Por otro lado, se han realizado los despliegues en Heroku. Heroku es una Plataforma como Servicio (Platform-as-a-Service; PaaS), conocida por ser una de las primeras plataformas de computación en la nube [45]. Esta plataforma soporta diferentes lenguajes, como NodeJS, Ruby, Java, PHP o Go, así como el despliegue de aplicaciones en contenedores [46]. Se ha elegido esta plataforma porque ofrece una capa gratuita que es ideal para entornos de desarrollo o preproducción.

3.3.3. Herramientas DevSecOps

Detección de secretos

El análisis de secretos se ha abordado en dos momentos del flujo de desarrollo: (1) como un *hook* antes de realizar el *commit* de los cambios, mediante la herramientas Husky y detect-secrets-launcher, y (2) como un *job* en el *workflow* de GitHub Actions.

Husky es una herramienta que funciona como un paquete de NPM que permite automatizar la ejecución de determinadas acciones antes de realizar acciones como el *commit* de cambios en nuestro código, o la subida de estos cambios al repositorio de código [53]. La principal ventaja de ejecutar una herramienta que detecte los secretos antes de realizar el *commit*, es que, en caso de que haya alguna filtración, no se completará el *commit* y, por tanto, el secreto no formará parte del historial de cambios del repositorio. La herramienta de detección de secretos automatizada en este paso ha sido detect-secrets-launcher [54], un *wrapper* en JavaScript de *detect-secrets* [55]. La herramienta detect-secrets realiza la búsqueda de secretos en base a palabras clave, expresiones regulares, y búsquedas por entropía.

La herramienta empleada para el escaneo de secretos en el *workflow* ha sido TruffleHog. [56]. Esta herramienta hace comprobaciones similares a las de detect-secrets a la hora de buscar secretos en el código, pero no solo escanea el directorio de código, sino que además escanea los últimos cambios realizados en el código. De esta manera evita que se publiquen los secretos en el repositorio de código si

detecta alguno en el historial de cambios. Además, permite configurar de forma más sencilla nuevas expresiones regulares para la detección de secretos y excluir ficheros del escaneo.

Pruebas Estáticas de la Seguridad de la Aplicación (SAST)

Al igual que en la gestión de secretos, se han realizados pruebas SAST tanto en un *hook* antes de realizar el *commit* de los cambios en el código, como en el *workflow* de GitHub Actions. El *hook pre-commit* se ha automatizado también con Husky. Como herramienta SAST a ejecutar en este paso se ha empleado ESlint con el *plugin* eslint-security. ESlint es una herramienta empleada para analizar el código de aplicaciones tanto JavaScript como TypeScript en busca de *code smells*, de malas prácticas o de *bugs*, y para forzar un estilo de código concreto en un proyecto [57]. Gracias al *plugin* eslint-security, ESlint permite realizar un análisis estático de la seguridad en base a reglas preestablecidas. Estas reglas permiten detectar varias vulnerabilidades: inyecciones SQL, uso de funciones poco seguras como *eval* y/o malas prácticas como el uso de la función *require* con una variable en lugar de con un valor literal [58].

En el *workflow* se han incluido dos tipos de pruebas estáticos de la seguridad, una de composición del código (SCA) y otra SAST. Para el análisis SCA se ha utilizado npm audit, una funcionalidad de NPM. El comando npm audit permite analizar las dependencias instaladas y detectar los paquetes que presentan vulnerabilidades comprobándolas con una gran base de datos [59].

Para las pruebas SAST se han empleado las herramientas CodeSQL y Semgrep. Por un lado, CodeQL es una herramienta multilenguaje de análisis semántico de código que proporciona GitHub. Esta herramienta ejecuta un tipo de análisis denominado análisis de variantes, el cual toma una vulnerabilidad conocida como origen para encontrar problemas similares en el código. El análisis de esta herramienta se divide en tres fases. En la primera se prepara el código creando una base de datos, en la segunda se ejecutan peticiones CodeQL contra la base de datos, y por último, se interpretan los resultados determinando las vulnerabilidades encontradas [60]. Por otro lado, Semgrep también es una herramienta SAST multilenguaje, la cual ejecuta un análisis de seguridad del código en base a una serie de reglas preestablecidas. Estas reglas se definen como trozos de código vulnerables, de forma que si algo similar a lo que establece esta regla se encuentra en la base de código a analizar, se puede considerar que existe una vulnerabilidad [61]. Las ventajas que presenta esta herramienta son la facilidad con la que se pueden definir nuevas reglas, la gran base de reglas que proporcionan inicialmente para cada lenguaje, y la interfaz web que permite ver los reportes de las ejecuciones de la herramienta.

Pruebas Dinámicas de la Seguridad de la Aplicación (DAST)

La herramienta DAST empleada ha sido StackHawk. Esta herramienta está compuesta por dos partes: el escáner de vulnerabilidades HawkScan y StackHawk Platform. El escáner HawkScan utiliza como base OWASP ZAP, mejorando la automatización, los resultados, y las integraciones de este [62]. OWASP ZAP realiza una búsqueda de las diferentes URLs y de los recursos disponibles en la web, para luego ejecutar diferentes ataques conocidos contra los objetivos especificados [63]. Estos ataques van desde inyección de cabeceras hasta inyecciones SQL, pasando por intentos de denegación de servicio, entre otras pruebas. La gran ventaja de StackHawk frente OWASP ZAP es su orientación a entornos CI/CD, que mejora su integración en *pipelines*.

Escaneo de la Infraestructura

Como herramienta de escaneo de infraestructura se ha utilizado Trivy. Esta herramienta es un escáner de vulnerabilidades de imágenes y de otros artefactos que detecta las vulnerabilidades tanto para el sistema operativo sobre el que se ha construido la imagen, como para los paquetes y otras dependencias empleadas en la creación de la imagen. Asimismo, Trivy está pensada para su fácil integración en *pipelines* de integración continua y provee unos resultados fácilmente comprensibles [64].

4. Implementación

En este apartado se detalla la implementación de la *pipeline* diseñada el apartado 3.2. La estructura de paratados se corresponde por lo tanto con el orden de los *jobs* descritos en el flujo de trabajo. Podemos ver el código completo tanto en el anexo A.1 o en el repositorio de código público en GitHub https://github.com/JYisus/bestreads/blob/master/.github/workflows/ci.yaml.

4.1. Análisis de secretos

Configuración detección secretos con Husky

El primer paso para configurar Husky ha sido instalarlo como dependencia de desarrollo del proyecto. A continuación, se ha configurado el *script* npm prepare para que instale de forma global la dependencia. Por último, se genera el fichero *pre-commit* en el directorio *.husky* que llevará a cabo dos acciones: ejecutar *detect-secrets-launcher*, y a continuación ejecuta el *linter*. Pueden verse comandos ejecutados en la Figura 4.5. Se está llamando a npm run find:secrets y a npm run lint. Estos *script* se han definido en el fichero package.json tal y como se ve en la figura 4.6.

```
npm install --save-dev husky

npm set-script prepare "husky install" && npm run prepare

npx husky add .husky/pre-commit "npm run find:secrets && npm run lint"
```

Figura 4.5: *Job* análisis de secretos

```
"find:secrets":
"npx detect-secrets-launcher $(git ls-files) --exclude-secrets='password'
--exclude-files package.json"
,
    "lint": "npx eslint .",
```

Figura 4.6: *Scripts* detección de secretos y *lint* en package.json

Configuración detección de secretos en el workflow

Como herramienta de escaneo de secretos en el *workflow* de GitHub Actions se ha utilizado TruffleHog. A esta herramienta se le ha activado el escaneo de expresiones regulares y se le ha indicado una profundidad máxima de 5, lo que implica que escaneará los últimos 5 *commits* en busca de posibles secretos que no se hayan detectado en el *pre-commit*. Además, con la opción -x se le indica que en el fichero .trufflehogignore se encuentra una lista de fichero y directorios que se han de ignorar en el escaneo. Entre los directorios a ignorar están los tests, el directorio node_modules y el fichero package-lock.json.

TruffleHog se instala mediante el instalador de paquetes de Python pip, por lo que, para evitar problemas de versiones, se ha optado por emplear su imagen de Docker. La Figura 4.7 muestra la configuración completa del *job* y en la Figura 4.8 se muestra un ejemplo de ejecución del *job* en el que se encuentra un secreto filtrado.

```
secrets-analysis:
  runs-on: ubuntu-20.04
  steps:
    - uses: actions/checkout@v2
    - name: Check secrets
      run:
docker run --rm -v "$(pwd):/proj" dxa4481/trufflehog --regex
 -x .trufflehogignore  --max_depth=5 file:///proj
```

Figura 4.7: *Job* análisis de secretos

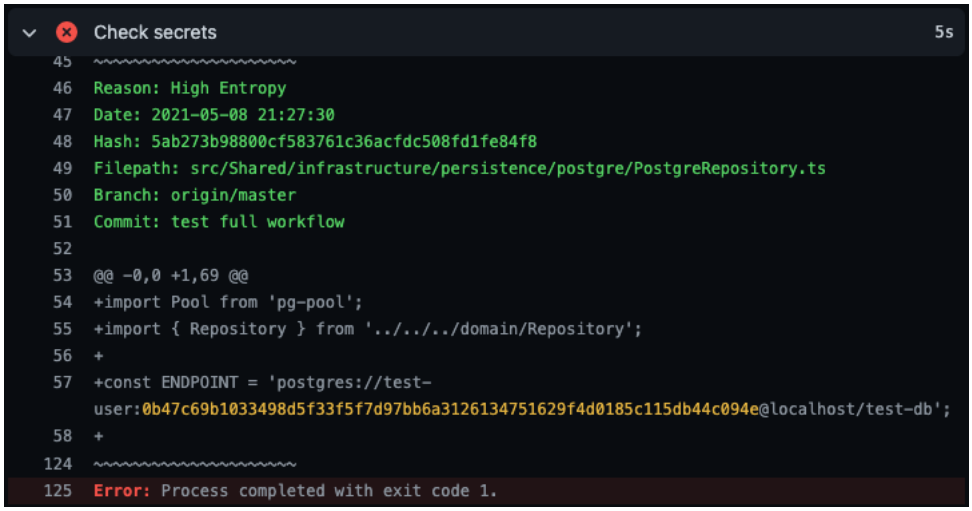

Figura 4.8: Ejemplo de *job* con secreto filtrado

4.2. Análisis estático de la seguridad de la aplicación

Configuración análisis estático Husky

Se ha modificado el *pre-commit* configurado con Husky, añadiendo la ejecución de Eslint, una herramienta usada principalmente para el análisis de estilo de código, con el fin de mantenerlo limpio y de acorde a ciertas reglas. Se ha configurado esta herramienta con el plugin *security* [58], la cual se encarga de añadir comprobaciones de seguridad.

Configuración análisis SCA en el workflow

El análisis de composición del software se realiza con la herramienta que proporciona Node Package Manager (NPM), el gestor de paquetes utilizado en el proyecto. Esta herramienta se ejecuta indicando mediante el argumento *audit*. Puede verse su definición en la Figura 4.12

Para probar su funcionamiento dentro del workflow, se ha instalado una paquete vulnerable, en este caso la versión 1.0.6 de Netmask [68]. En esta herramienta, utilizada para trabajar con direcciones IP, el día 19 de marzo de 2021 se encontró una vulnerabilidad [69] causada por una incorrecta comprobación de cadenas octales, la cual permitía a atacantes remotos no autenticados realizar ataques *Server-Side Request Forgery (SSRF)*, *Remote File Inclusion (RFI)*, y *Local File Inclusion (LFI)*. [70].

Se muestra en la Figura 4.9 el momento en el que el escaneo de dependencias detecta esta vulnerabilidad en el *workflow*.

Figura 4.9: Detección de paquete vulnerable con *npm audit*

Configuración análisis SAST en el workflow

En cuanto al análisis SAST, se han utilizado dos herramientas: CodeQL y Semgrep. La primera la ofrece GitHub como una herramienta de seguridad avanzada del repositorio. Se activa desde el panel de seguridad del repositorio, en la pestaña *Code scanning alerts*. Aquí se pueden ver una serie de *workflows* disponibles en el *marketplace* de GitHub, siendo el primero que se ofrece CodeQL Analysis (ver Figura 4.10). Al activar este *workflow*, se genera un archivo .yaml con las configuración necesaria para la ejecución del mismo en el directorio .github del repositorio del proyecto. Esta configuración se ha adaptado a nuestro caso de uso y se ha incluido como parte del *job* static-analysis en lugar de dedicar un *workflow* para tener las herramientas de análisis estático de la seguridad agrupadas.

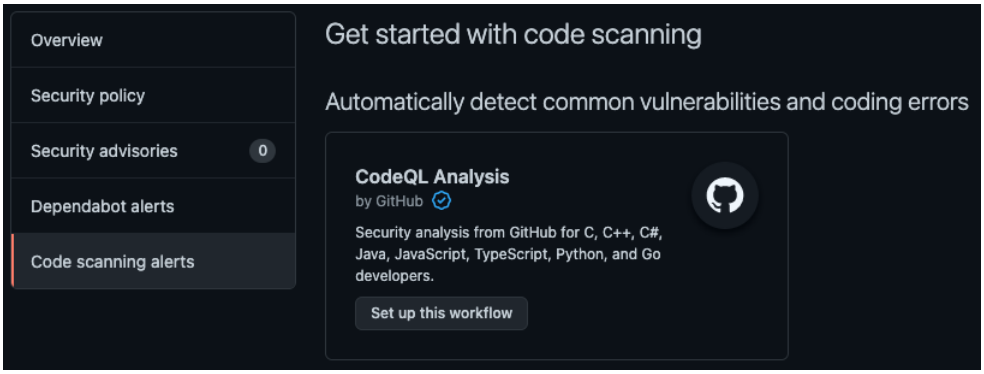

Figura 4.10: Panel *Code scanning alerts* del repositorio

Semgrep es también una herramienta de análisis de código, similar a CodeQL, con la diferencia de que, gracias a su interfaz web, se pueden configurar fácilmente normas adicionales de seguridad, las cuales permiten comprobaciones específicas en esta fase, además de ofrecer un *dashboard* donde observar los resultados de los análisis realizados (ver Figura 4.11). La integración de esta herramienta en el *workflow* se ha realizado mediante la acción que los propios creadores de Semgrep han desarrollado, y puede verse en la Figura 4.12. Como parte de la configuración de esta herramienta, se ha creado una nueva política de seguridad, esto es, un conjunto de reglas que utiliza la herramienta para llevar a cabo el escaneo. Las reglas que se han añadido a esta ha sido las referentes a JavaScript y TypeScript ofrecidas como reglas base de Semgrep que no estaban orientadas a desarrollo *front-end* [72], las que implicaban comprobaciones sobre el uso de Json Web Tokens [73] ofrecidas también por la herramienta, y, por último, un conjunto de reglas adaptadas de la herramienta NodeJSScan, orientadas al desarrollo en NodeJS [74].

Finding	Introduced	Project	Ref	Location	Severity	Status
writable-filesystem-service #212925 from docker-compose	an hour ago by semgrep-dev[bot]	JYisus/bestreads	Branch: master	docker-compose.yml:3	WARNING	OPEN
no-new-privileges #212924 from docker-compose	an hour ago by semgrep-dev[bot]	JYisus/bestreads	Branch: master	docker-compose.yml:12	WARNING	OPEN
no-new-privileges #212923 from docker-compose	an hour ago by semgrep-dev[bot]	JYisus/bestreads	Branch: master	docker-compose.yml:3	WARNING	OPEN
detect-non-literal-require #212922 from security-audit	an hour ago by semgrep-dev[bot]	JYisus/bestreads	Branch: master	src/app/routes/index.ts:6	WARNING	OPEN
writable-filesystem-service #212921 from docker-compose	an hour ago by semgrep-dev[bot]	JYisus/bestreads	Branch: master	docker-compose.yml:12	WARNING	OPEN

Figura 4.11: Resultados de análisis en *dashboard* Semgrep

Figura 4.12: *Job* análisis estático de la seguridad

En la Figura 4.13 puede verse el resultado de un *worflow* que falla por haberse detectado código vulnerable durante el escáner estático realizado con Semgrep. Puede observarse en el *log*, que el código analizado se está utilizando las función hash SHA-1, lo que conlleva a una debilidad calificada como *CWE-327: Uso de un algoritmo criptográfico poco seguro* [75]. Además, se está encontrando otra alerta en lo referente al uso de la función *require* empleando una variable. Esto podría ser problemático si un usuario pudiese manipular esta variable, pues podría ejecutar código remoto, pero en este caso se trata de una situación controlada. Otras vulnerabilidades o debilidades que puede prevenir el uso de herramientas de escaneo estático son: inyecciones de código SQL (CWE-89) [76], inyecciónes de código NoSQL (CWE-943) [78], inyección de cabeceras (CWE-644) [80] o ataques de *Cros-site scripting* (CWE-79) [81].

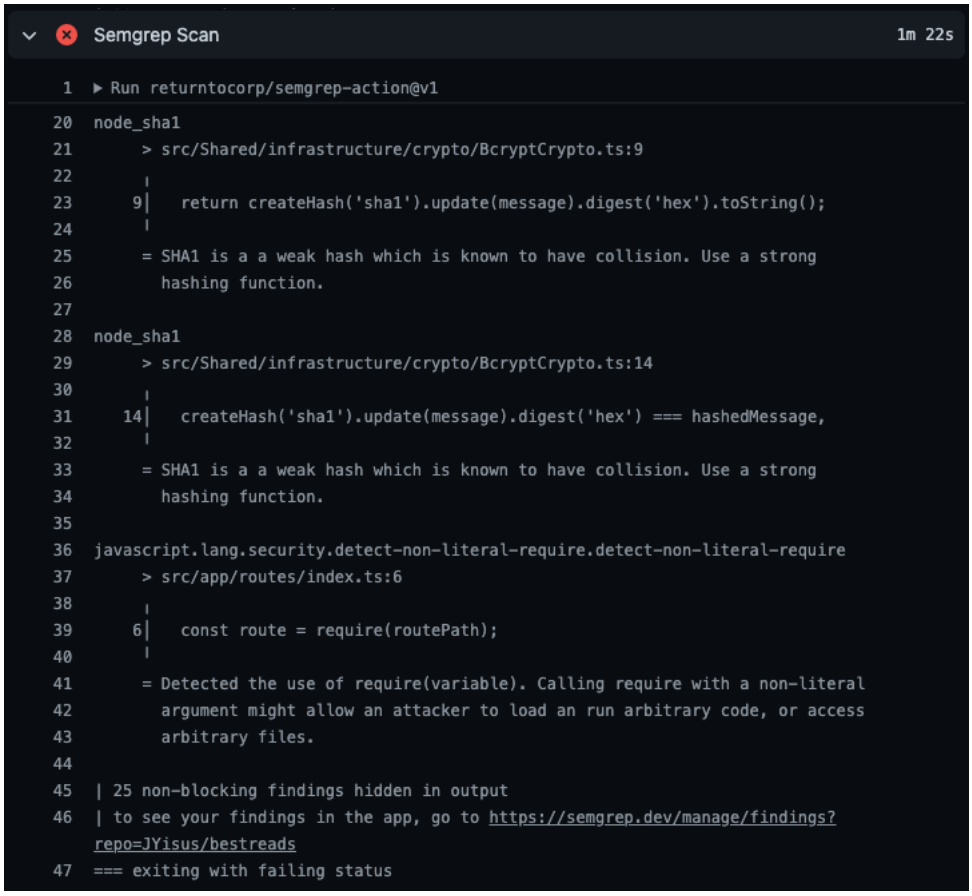

Figura 4.13: Resultados GitHub Actions Semgrep

4.3. Tests

La ejecución de pruebas automatizadas es un paso fundamental en toda *pipeline* CI/CD. En este caso, se ha optado por la ejecución de tests de integración, pues estos implican una mayor complejidad al probarse las conexiones que realiza la aplicación con otros componentes de la arquitectura, en este caso, con la base de datos. Para levantar una base de datos contra la que se ejecutarán los tests, se hace uso de los servicios o *services* de GitHub Actions. El funcionamiento y la definición de estos es similar a los contenedores de Docker, puede verse en la Figura 4.14 como se ha realizado con la base de datos PostgreSQL.
Los pasos llevados a cabo para la ejecución de los tests una vez definido el servicio de la base de datos son:

- Configurar Node
- Instalar las dependencias en entorno CI

- Ejecutar el *script* que lanza los tests

```yaml
test:
  runs-on: ubuntu-20.04
  services:
    postgres:
      image: postgres
      env:
        POSTGRES_USER: postgres
        POSTGRES_PASSWORD: postgres
      ports:
        - 5432:5432
      options: >-
        --health-cmd pg_isready
        --health-interval 10s
        --health-timeout 5s
        --health-retries 5

  steps:
  - uses: actions/checkout@v2
  - name: Set up Node
    uses: actions/setup-node@v2
    with:
      node-version: '14'
  - name: Installing dependencies
    run: npm ci
  - name: Executing tests
    run: npm test
    env:
      POSTGRES_HOST: localhost
      POSTGRES_PORT: 5432
```

Figura 4.14: *Job* pruebas automatizadas

4.4. Construcción de la imagen

El objetivo de este *job* es construir la imagen Docker de la aplicación y subirla a un repositorio de imágenes, en este caso, el que ofrece el propio GitHub. Gracias

al campo *needs* podemos definir de qué *jobs* depende el que se está definiendo para su ejecución. En este caso, se ha establecido que, para proceder con la construcción de la imagen, se han de haber completado con éxito la detección de secretos, el análisis estático de código y los tests. Hay que tener presente que la ejecución de los *jobs* en GitHub Actions se realiza de forma paralela a no ser que definamos estas dependencias, y en este caso solo nos interesa generar y subir la nueva imagen si el workflow, hasta este punto, no ha encontrado ningún error.

```yaml
build:
  needs:  [secrets-analysis, static-analysis, test]
  runs-on: ubuntu-20.04
  steps:
    - uses: actions/checkout@v2
    - uses: docker/build-push-action@v2
      with:
        context: .
        tags: ghcr.io/jyisus/bestreads-backend:latest
        push: false
```

Figura 4.15: *Job* construcción sde la imagen

4.5. Escaneo de artefactos

El siguiente paso en la *pipeline* desarrollada es el escaneo de artefactos, en este caso, de la imagen Docker construida en el *job* anterior. Con este fin se ha utilizado Trivy. La ejecución se ha realizado mediante la ejecución con Docker de la herramienta 4.16.

```
infrastructure-scanning:
  needs: build
  runs-on: ubuntu-20.04
  steps:
    - name: Trivy Scanning
      run:
docker run --rm -v "/var/run/docker.sock:/var/run/docker.sock" aquasec/trivy
ghcr.io/jyisus/bestreads-backend:latest
```

Figura 4.16: *Job* escaneo de artefactos

4.6. Análisis dinámico de la seguridad de la aplicación

Como último *job* relacionado con la seguridad, se ha incluido la herramienta StackHawk, una herramienta DAST orientada principalmente a entornos CI/CD. Entre las ventajas que ofrece esta herramienta tenemos la facilidad de integración, la centralización de resultados de sus análisis a través de su *dashboard* accesible desde internet, y la facilidad de interpretación de estos resultados, facilitando así la corrección de los problemas encontrados por parte de los desarrolladores. Podemos ver un ejemplo de resultados de análisis en la Figura 4.17.

El primer paso para configurar esta herramienta ha sido crear una cuenta en esta herramienta. Se ha optado por una cuenta gratuita, pues es suficiente para las pruebas de este proyecto, pero se analizará su precio en las conclusiones de esta memoria. La cuenta gratuita permite integrar la herramienta en una aplicación, pudiendo seleccionarla directamente del repositorio GitHub. Una vez configurada la aplicación en StackHawk, obtendremos un identificador de aplicación, necesario para la configuración de la herramienta y un API KEY, el cual se guardará de forma segura en los secretos del repositorio, y configuramos el *job* en el *workflow* de GitHub Actions. Para esto, se opta por testear contra una aplicación ya desplegada en un entorno de desarrollo o producción, o directamente levantar un contenedor Docker en el mismo *job* y ejecutar StackHawk contra éste. La opción tomada es esta última, de forma que aseguraremos de que la aplicación es completamente segura antes de su despliegue. La configuración de ejecución de la herramienta se realiza en el fichero stackhawk.yml generado en el directorio raíz de la aplicación. En este se tiene que configurar tanto el ID de la aplicación,

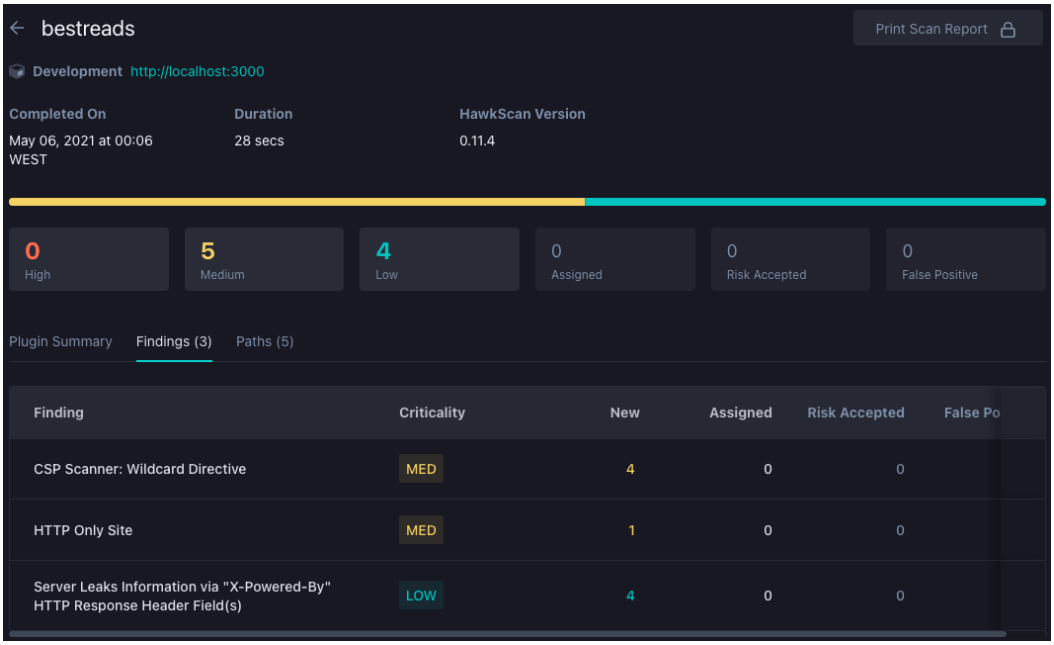

Figura 4.17: Análisis dinámico del código

como la URL contra la que realizar las pruebas y resto de opciones de ejecución de la misma. Por último, añadimos al *job* la acción que ejecuta StackHawk (ver Figura 4.18).

Las herramientas de análisis dinámico permiten encontrar vulnerabilidades de aplicaciones en ejecución, y que, por tanto, de otra forma sería complicado encontrar. Entre estas vulnerabilidades están las ya mencionadas anteriormente inyecciones de código SQL y los ataques de *Cross-site scripting*, además de otras como: denegaciones de servicio, corrupción de memoria, filtrado de información, o *local file inclusion* y *remote file inclusion*.

```
dast-analysis:
  needs: build
  runs-on: ubuntu-20.04
  steps:
    - uses: actions/checkout@v2
    - name: Run Bestreads api
      run:
docker run --rm --detach --publish 3000:3000 --name bestreads-backend
ghcr.io/jyisus/bestreads-backend:latest
    - name: HawkScan
      uses: stackhawk/hawkscan-action@v1.2
      with:
        apiKey: ${{ secrets.HAWK_API_KEY }}
```

Figura 4.18: Análisis dinámico del código

4.7. Subida de Imagen a Registro

Para la subida de la imagen al repositorio de GitHub se ha empleado la misma acción que para la construcción de la misma, esta vez configurando la opción *push* a *true*.

```
build:
  needs:  [secrets-analysis, static-analysis, test]
  runs-on: ubuntu-20.04
  steps:
    - uses: actions/checkout@v2
    - name: Log into registry
      run:
echo "${{ secrets.GITHUB_TOKEN }}" | docker login ghcr.io
-u ${{ github.actor }} --password-stdin
    - uses: docker/build-push-action@v2
      with:
        context: .
        tags: ghcr.io/jyisus/bestreads-backend:latest
        push: true
```

Figura 4.19: *Job* construcción sde la imagen

4.8. Despliegue de la Aplicación

El despliegue de la aplicación se realiza a la plataforma Heroku, y para ello se ha utilizado una acción desarrollada por la comunidad, cuya configuración simplemente necesita el correo de usuario de Heroku, la API KEY y el nombre de la aplicación. Puede observarse esta configuración en la Figura 4.20.

```yaml
deploy-pre:
  needs: push-image
  runs-on: ubuntu-20.04
  steps:
  - uses: actions/checkout@v2
  - uses: akhileshns/heroku-deploy@v3.12.12
    with:
      heroku_api_key: ${{ secrets.HEROKU_API_KEY }}
      heroku_app_name: bestreads-tfm
      heroku_email: ${{ secrets.HEROKU_EMAIL }}
      usedocker: true
```

Figura 4.20: Despliegue en Heroku

5. Resultados

Los resultados obtenidos en el proyecto han sido satisfactorios, habiéndose cumplido los objetivos planteados. El desarrollo de la *pipeline* de integración continua desarrollada en GitHub Actions ha demostrado la viabilidad de este servidor de integración continua para su uso tanto en proyectos pequeños como a nivel empresarial. La configuración de GitHub Actions se ha simplificado bastante gracias a las acciones, siendo esta una de las características más destacables frente a otros servidores de integración continua.

Aunque los resultados obtenidos son favorables, si se planteara una migración a GitHub Actions de empresas que estén utilizando algún otro sistema de integración continua, sería necesario analizar la viabilidad considerando diversos factores. En primer lugar, habría que considerar la complejidad de la solución en funcionamiento. Si se tratase de una empresa con un gran número de *pipelines* implementadas, la migración de estas a GitHub Actions conllevaría una gran inversión de tiempo, tanto en formación como en desarrollo. En segundo lugar, habría que considerar si la empresa trabaja con un gestor de repositorios *on-premise*. En caso de que sí, para disponer de esta misma funcionalidad en GitHub, sería necesario contratar su plan para empresas, lo que conllevaría un coste adicional por usuario, frente a otras plataformas como por ejemplo GitLab, que incluye esta funcionalidad de forma gratuita. Si por el contrario, se tratase de una empresa que tenga contratados los planes para equipos o para empresas de GitLab, las migración a GitHub podría suponer un gran ahorro anual, pues los planes tienen un precio más reducido. En el Cuadro 5.20 se reflejan las comparaciones de los costes anuales que supondrían los planes de equipo y de empresa de GitHub y de GitLab para compañías con 5, 10 y 20 desarrolladores.

Cuadro 5.20: Comparativa del coste anual GitHub vs GitLab

	GitHub	**GitLab**
Equipo		
5 desarrolladores	196.8€	933.6€
10 desarrolladores	393.6€	1867.2€
20 desarrolladores	787.2€	3734.4€
Empresa		
5 desarrolladores	1020€	4872€
10 desarrolladores	2040€	9744€
20 desarrolladores	4080€	19488€

6. Conclusiones y Futuros Trabajos

Este trabajo se ha llevado a cabo con ciertas limitaciones que deben tenerse en cuenta. La principal limitación reside en no haber implementado el producto obtenido en un entorno real. La aplicación desarrollada para simular una aplicación real no tiene todas las funcionalidades que podría tener una aplicación en producción, puesto que el implementar una aplicación más compleja no entraba dentro de los alcances del proyecto. De realizarse con un software real, mejorarían las conclusiones obtenidas, pues ayudaría a detectar las limitaciones tanto del producto como de la plataforma. Además, en un entorno real en el que varios desarrolladores estuvieran trabajando en una aplicación y realizasen varios cambios e integraciones al día por medio del producto desarrollado, se podría tener un *feedback* de este, que ayudaría a conocer los cuellos de botella y los aspectos a mejorar. Otro aspecto que ha supuesto una limitación ha sido la ausencia de presupuesto, optándose a lo largo de este proyecto por herramientas gratuitas. Muchas de estas, como GitHub Actions, ofrecen sus servicios de forma gratuita para proyectos públicos. Sin embargo, de haber contado con presupuesto, se podrían haber realizado algunas mejoras, como disponer de máquinas propias en la nube.

A pesar de las limitaciones mencionadas, este trabajo consigue implementar una *pipeline* de integración continua en la cual se realizan pruebas de seguridad del código, lo que supone una contribución al área de la seguridad empresarial, con una aplicación práctica de gran utilidad en empresas que quieran incorporar la seguridad en todo el ciclo de desarrollo del software de acuerdo con los principios de la cultura DevSecOps.
Futuros trabajos que pretendan ampliar el alcance de este proyecto, deberían optimizar la *pipeline* desarrollada, ya que actualmente esta tarda una media de 9 minutos en ejecutarse, cifra que puede mejorarse haciendo uso del sistema de cachés de GitHub Actions. Además, podrían contemplar en uso de *runners* autoalojadas. Esto conllevaría una mejora del rendimiento y de la seguridad, pues el proceso se ejecutaría en servidores propios.

Además, conllevaría una mejora notoria de la solución propuesta incluir herramientas de gestión de *logs* y de monitorización, ya que permitirían encontrar errores no detectados durante la integración continua, problemas de rendimiento en determinadas funcionalidades, o usos inadecuados. Las herramientas de gestión de logs y de monitorización serían también gran fuente de datos del uso que los usuarios le dieran a la aplicación, con lo que proporcionarían información sobre aspectos a mejorar en futuras actualizaciones. Incorporando estas mejoras, el producto final podría aplicarse en un entorno real, mejorando la seguridad dentro del ciclo de desarrollo de software siguiendo la cultura DevSecOps.

Referencias

[1] Miller, M. (2020, 16 abril). *FBI sees spike in cyber crime reports during coronavirus pandemic.*TheHill. https://thehill.com/policy/cybersecurity/493198-fbi-sees-spike-in-cyber-crime-reports-during-coronavirus-pandemic

[2] *What are microservices?*(s. f.). microservices.io. Recuperado 8 de marzo de 2021, de `https://microservices.io/`

[3] Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2016). *DevOps.* IEEE Software, 33(3), 94–100.

[4] Contrast Security. (2020, noviembre). *The state of DevSecOps Report.*(s. f.). microservices.io. Recuperado 8 de marzo de 2021, de `https://www.contrastsecurity.com/hubfs/DocumentsPDF/The-State-of-DevSecOps_Report_Final.pdf`

[5] MacDonald, N., Head, I. (2016) *DevSecOps: How to Seamlessly Integrate Security Into DevOps.* Technical report, Gartner.

[6] Mohan, V., Othmane, L.B. (2016, agosto) *Secdevops: is it a marketing buzzword? - mapping research on security in devops.* 11th International Conference on Availability, Reliability and Security (ARES), pp. 542–547

[7] JetBrains. (2020, junio). *The State of Developer Ecosystem in 2020 Infographic.* JetBrains: Developer Tools for Professionals and Teams. https://www.jetbrains.com/lp/devecosystem-2020/

[8] OpenJS Foundation. (s. f.). *Acerca de Node.js.* Node.js. Recuperado 20 de marzo de 2021, de https://nodejs.org/es/about/

[9] GitHub. (s. f.). *GitHub Actions.* Recuperado 28 de febrero de 2021, de `https://github.com/features/actions`

[10] Allspaw, J., & Hammond, P. (2009, junio 22–24). *10+ Deployes per Day: Dev and Ops Cooperation at Flickr [Talk].* O'Reilly Velocity Conference, San José, California. https://www.youtube.com/watch?v=LdOe18KhtT4

[11] Fowler, M. (2006, 1 mayo). *Continuous Integration.* martinfowler.com. https://www.martinfowler.com/articles/continuousIntegration.html

[12] J. Humble, & D. Farley (2011). *Continuous Delivery* Addison- Wesley.

[13] RedHat. (s. f.). *¿Qué son la integración/distribución continuas (CI/CD)?* Recuperado 1 de marzo de 2021, de `https://www.redhat.com/es/topics/devops/what-is-ci-cd`

[14] GitKraken. (2020). *DevOps Tools Report 2020 — GitKraken Resources.* GitKraken.com. https://www.gitkraken.com/reports/devops-report-2020

[15] Docker. (s. f.). *Docker.* Recuperado 27 de febrero de 2021, de `https://www.docker.com/`

[16] Podman. (s. f.). *Podman.* Recuperado 27 de febrero de 2021, de `https://podman.io/`

[17] Kubernetes. (s. f.). *What is Kubernetes?* Recuperado 27 de febrero de 2021, de `https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/`

[18] The OWASP Foundation. (s. f.). *OWASP DevSecOps Guideline.* OWASP. Recuperado 24 de marzo de 2021, de https://owasp.org/www-project-devsecops-guideline/

[19] Meli, M., McNiece, M. R., & Reaves, B. (2019). *How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories.* Proceedings 2019 Network and Distributed System Security Symposium. https://doi.org/10.14722/ndss.2019.23418

[20] Siddarth, S. (2020, 17 julio). *Herramientas open source para adoptar DevSecOps.* Claranet. https://www.claranet.es/blog/herramientas-open-source-para-adoptar-devsecops

[21] Virmani, M. (2015). Understanding DevOps & bridging the gap from continuous integration to continuous delivery. *Fifth International Conference on the Innovative Computing Technology (INTECH 2015).* Published. https://doi.org/10.1109/intech.2015.7173368

[22] de França, B. B. N., Jeronimo, H., & Travassos, G. H. (2016). Characterizing DevOps by Hearing Multiple Voices. *Proceedings of the 30th Brazilian Symposium on Software Engineering - SBES '16.* Published. https://doi.org/10.1145/2973839.2973845

[23] Jabbari, R., bin Ali, N., Petersen, K., & Tanveer, B. (2016). What is DevOps? *Proceedings of the Scientific Workshop Proceedings of XP2016.* Published. https://doi.org/10.1145/2962695.2962707

[24] Watts, S. (2019, 29 marzo). *A Brief History of DevOps.* BMC Blogs. https://www.bmc.com/blogs/devops-history/

[25] Debois, P. (s. f.). *About devopsdays.* DevOpsDays. Recuperado 18 de mayo de 2021, de https://devopsdays.org/about/

[26] Akshaya, H. L., Nisarga Jagadish, S., Bidya, J., & Veena, K. (2015). A Basic Introduction to DevOps Tools. *International Journal of Computer Science and Information Technologies, 6(3).* http://ijcsit.com/docs/Volume%206/vol6issue03/ijcsit2015060382.pdf

[27] Beck, K., Fowler, M., Martin, R. C., Beedle, M., Cockburn, A., Cunningham, W., Thomas, D., Mellor, S., Schwaber, K., Sutherland, J., Bennekum, A. V., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., & Marick, B. (2001, 13 febrero). *Principios del Manifiesto Ágil.* Agile Manifesto. http://agilemanifesto.org/iso/es/principles.html

[28] Shackleford, D. (2016, 8 marzo). *A DevSecOps Playbook.* SANS. https://www.sans.org/webcasts/devsecops-playbook-101472

[29] Amazon Web Services. (2016, 9 noviembre). *Introduction to DevSecOps on AWS*. https://www.slideshare.net/AmazonWebServices/introduction-to-devsecops-on-aws-68522874

[30] GitLab. (s. f.). *Secret Detection.* GitLab Docs. Recuperado 20 de mayo de 2021, de https://docs.gitlab.com/ee/user/application_security/secret_detection/

[31] GitHub. (s. f.). *About secret scanning*. GitHub Docs. Recuperado 20 de mayo de 2021, de https://docs.github.com/en/code-security/secret-security/about-secret-scanning

[32] Adkins, H., Beyer, B., Blankinship, P., Lewandowski, P., Oprea, A., & Stubblefield, A. (2020). *Building Secure and Reliable Systems: Best Practices for Designing, Implementing, and Maintaining Systems (Illustrated ed.).* O'Reilly Media. https://sre.google/books/building-secure-reliable-systems/

[33] Peterson, J. (2020, noviembre 19). *Software Composition Analysis Explained.* WhiteSource. https://www.whitesourcesoftware.com/resources/blog/software-composition-analysis/

[34] Gartner. (s.f.). *Static Application Security Testing (SAST).* Recuperado el 28 de febrero de 2021 de https://www.gartner.com/en/information-technology/glossary/static-application-security-testing-sast

[35] Veracode. (s. f.). *DAST Test.* Recuperado 21 de mayo de 2021, de *https://www.veracode.com/security/dast-test*https://www.veracode.com/security/dast-test

[36] Hsu, T. (2018). *Hands-On Security in DevOps: Ensure continuous security, deployment, and delivery with DevSecOps.* Packt Publishing. https://www.packtpub.com/product/hands-on-security-in-devops/9781788995504

[37] Artac, M., Borovssak, T., Di Nitto, E., Guerriero, M., & Tamburri, D. A. (2017). DevOps: Introducing Infrastructure-as-Code. *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C).* Published. https://doi.org/10.1109/icse-c.2017.162

[38] *What is Docker?* (s. f.). opensource.com. Recuperado 25 de mayo de 2021, de https://opensource.com/resources/what-docker

[39] Darshan, N. (2021, 17 abril). *Docker Image Vulnerabilities — Trivy Image Scan Guide.* Cloud Training Program. https://k21academy.com/docker-kubernetes/docker-image-vulnerabilities/

[40] Wells, D. (1999). *Continuous Integration.* Extreme Programming. http://www.extremeprogramming.org/rules/integrateoften.html

[41] Fowler, M. (2000, 10 septiembre). *Continuous Integration (original version).* martinfowler.com. https://www.martinfowler.com/articles/originalContinuousIntegration.html

[42] Wells, D. (1999b). *Dedicated Release Computer.* Extreme Programming. http://www.extremeprogramming.org/rules/dedicated.html

[43] SonarCube. (s. f.). *SAST Testing — Code Security & Analysis Tools.* Recuperado 1 de marzo de 2021, de `https://www.sonarqube.org/features/security/`

[44] Rahman, A. A. U., Helms, E., Williams, L., & Parnin, C. (2015). Synthesizing Continuous Deployment Practices Used in Software Development. *2015 Agile Conference.* Published. https://doi.org/10.1109/agile.2015.12

[45] Heroku. (s. f.). *Cloud Application Platform*. Recuperado 30 de mayo de 2021, de https://www.heroku.com/

[46] Heroku. (s. f.-b). *Deploying with Docker.* Heroku Dev Center. Recuperado 30 de mayo de 2021, de https://devcenter.heroku.com/categories/deploying-with-docker

[47] Humble, J., Read, C., & North, D. (2006). The Deployment Production Line. *AGILE 2006 (AGILE'06).* Published. https://doi.org/10.1109/agile.2006.53

[48] Fowler, M. (2013, 30 mayo). *ContinuousDelivery.* martinfowler.com. https://martinfowler.com/bliki/ContinuousDelivery.html

[49] GitLab. (s. f.-a). *European-Space-Agency.* Recuperado 28 de mayo de 2021, de https://about.gitlab.com/customers/european-space-agency/

[50] GitHub. (s. f.-b). *Events that trigger workflows.* GitHub Docs. Recuperado 28 de mayo de 2021, de https://docs.github.com/en/actions/reference/events-that-trigger-workflows

[51] GitHub. (s. f.-a). *About actions.* GitHub Docs. Recuperado 28 de mayo de 2021, de https://docs.github.com/en/actions/creating-actions/about-actions

[52] GitHub. (s. f.-d). *Workflow syntax for GitHub Actions.* GitHub Docs. Recuperado 30 de mayo de 2021, de https://docs.github.com/en/actions/reference/workflow-syntax-for-github-actions

[53] Typicode. (s. f.). *Husky - Git hooks.* Recuperado 23 de mayo de 2021, de https://typicode.github.io/husky/

[54] *npm: detect-secrets.* (s. f.). Npm. Recuperado 23 de mayo de 2021, de https://www.npmjs.com/package/detect-secrets

[55] Yelp. (s. f.). *Yelp/detect-secrets.* GitHub. Recuperado 23 de mayo de 2021, de https://github.com/Yelp/detect-secrets

[56] TuffleSecurity. (s. f.). *TruffleHog.* Truffle Security. Recuperado 23 de mayo de 2021, de https://trufflesecurity.com/trufflehog

[57] Hernandez, M. (2021, 26 enero). *¿Qué es Linting y ESLint? ¿Cómo empezar?* freeCodeCamp.org. https://www.freecodecamp.org/espanol/news/que-es-linting-y-eslint/

[58] npm: eslint-plugin-security. (2017, 13 junio). Npm. https://www.npmjs.com/package/eslint-plugin-security

[59] NPM. (s. f.). *npm-audit.* NPM Docs. Recuperado 24 de mayo de 2021, de https://docs.npmjs.com/cli/v7/commands/npm-audit/https://docs.npmjs.com/cli/v7/commands/npm-audit/

[60] GitHub. (s. f.-b). *About CodeQL.* CodeQL. Recuperado 24 de mayo de 2021, de https://codeql.github.com/docs/codeql-overview/about-codeql/

[61] Semgrep. (s. f.). *Semgrep Docs.* Recuperado 24 de mayo de 2021, de https://semgrep.dev/docs/

[62] StackHawk. (s. f.). *HawkDocs.* StackHawk Docs. Recuperado 25 de mayo de 2021, de https://docs.stackhawk.com/

[63] Gokte, S. (2017, 5 diciembre). *An intro to OWASP Zed Attack Proxy.* Srijan. https://www.srijan.net/blog/intro-owasp-zed-attack-proxy

[64] AquaSecurity. (s. f.). *Trivy.* Recuperado 25 de mayo de 2021, de https://aquasecurity.github.io/trivy/v0.18.3/

[65] Weerasinghe, M. (2019, 24 diciembre). *NodeJS Security Tools - Manjula Weerasinghe.* Medium. https://medium.com/@manjula.aw/nodejs-security-tools-de0d0c937ec0

[66] Gartner. (s. f.). *Definition of Dynamic Application Security Testing (DAST).* Recuperado 1 de marzo de 2021, de https://www.gartner.com/en/information-technology/glossary/dynamic-application-security-testing-dast

[67] Peterson, J. (2020, 30 julio). *Dynamic Application Security Testing: DAST Basics.* WhiteSource. https://resources.whitesourcesoftware.com/blog-whitesource/dast-dynamic-application-security-testing

[68] *npm: netmask.* (2016, 30 mayo). npm. https://www.npmjs.com/package/netmask/v/1.0.6

[69] *CVE-2021-28918.* (2021, 19 marzo). CVE Mitre. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-28918

[70] Arghire, I. (2021, 29 marzo). *Vulnerability in «netmask» npm Package Affects 280,000 Projects* SecurityWeek. https://www.securityweek.com/vulnerability-netmask-npm-package-affects-280000-projects

[71] The Node Security Platform. (s. f.). *nodesecurity/eslint-plugin-security.* GitHub. Recuperado 17 de abril de 2021, de https://github.com/nodesecurity/eslint-plugin-security

[72] Hardaway, G. (s. f.). *Javascript.* Recuperado 5 de mayo de 2021, de https://semgrep.dev/p/javascript

[73] Hardaway, G. (s. f.). *JWT.* Recuperado 5 de mayo de 2021, de https://semgrep.dev/p/jwt

[74] Abraham, A. (s. f.). *NodeJSScan.* Semgrep. Recuperado 5 de mayo de 2021, de https://semgrep.dev/p/nodejsscan

[75] *CWE - CWE-327: Use of a Broken or Risky Cryptographic Algorithm.* (2006, 19 julio). CWE. https://cwe.mitre.org/data/definitions/327.html

[76] *CWE - CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection').* (2006, 19 julio). CWE Mitre. https://cwe.mitre.org/data/definitions/89.html

[77] *CWE - CWE-943: Improper Neutralization of Special Elements in Data Query Logic.* (2014, 19 junio). CWE Mitre. https://cwe.mitre.org/data/definitions/943.html

[78] *CWE - CWE-943: Improper Neutralization of Special Elements in Data Query Logic.* (2014, 19 junio). CWE Mitre. https://cwe.mitre.org/data/definitions/943.html

[79] *CWE - CWE-644: Improper Neutralization of HTTP Headers for Scripting Syntax.* (2008, 30 enero). CWE Mitre. https://cwe.mitre.org/data/definitions/644.html

[80] *CWE - CWE-644: Improper Neutralization of HTTP Headers for Scripting Syntax.* (2008, 30 enero). CWE Mitre. https://cwe.mitre.org/data/definitions/644.html

[81] *CWE - CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting').* (2006, 19 julio). CWE Mitre. https://cwe.mitre.org/data/definitions/79.html

A. Anexos

A.1. Código del workflow

```
name: DevSecOps CI/CD

on: [push, pull_request]

jobs:
  secrets-analysis:
    runs-on: ubuntu-20.04
    steps:
      - uses: actions/checkout@v2
      - name: Check secrets
        run: docker run --rm -v
        "$(pwd):/proj" dxa4481/trufflehog --regex -x
        .trufflehogignore  --max_depth=1 file:///proj

  static-analysis:
    runs-on: ubuntu-20.04
    steps:
    - uses: actions/checkout@v2
    - name: Scanning dependencies
      run: npm audit

    - name: Semgrep Scan
      uses: returntocorp/semgrep-action@v1
      with:
        auditOn: push
        publishToken: ${{ secrets.SEMGREP_APP_TOKEN }}
        publishDeployment: 1041

    - name: Initialize CodeQL
      uses: github/codeql-action/init@v1
      with:
        languages: 'javascript'
    - name: Autobuild
      uses: github/codeql-action/autobuild@v1
    - name: Perform CodeQL Analysis
      uses: github/codeql-action/analyze@v1
```

```
test:
  runs-on: ubuntu-20.04
  services:
    postgres:
      image: postgres
      env:
        POSTGRES_USER: postgres
        POSTGRES_PASSWORD: postgres
      ports:
        - 5432:5432
      options: >-
        --health-cmd pg_isready
        --health-interval 10s
        --health-timeout 5s
        --health-retries 5
  steps:
  - uses: actions/checkout@v2
  - name: Set up Node
    uses: actions/setup-node@v2
    with:
      node-version: '14'
  - name: Installing dependencies
    run: npm ci
  - name: Executing tests
    run: npm test
    env:
      POSTGRES_HOST: localhost
      POSTGRES_PORT: 5432

build:
  needs:  [secrets-analysis, static-analysis, test]
  runs-on: ubuntu-20.04
  steps:
    - uses: actions/checkout@v2
    - uses: docker/build-push-action@v2
      with:
        context: .
        tags: ghcr.io/jyisus/bestreads-backend:latest
        push: false

infrastructure-scanning:
```

```
  needs: build
  runs-on: ubuntu-20.04
  steps:
    - name: Trivy Scanning
      run: docker run --rm -v
      "/var/run/docker.sock:/var/run/docker.sock"
      aquasec/trivy ghcr.io/jyisus/bestreads-backend:latest

dast-analysis:
  needs: build
  runs-on: ubuntu-20.04
  steps:
    - uses: actions/checkout@v2
    - name: Run Bestreads api
      run: docker run --rm --detach --publish 3000:3000
      --name bestreads-backend
      ghcr.io/jyisus/bestreads-backend:latest
    - name: HawkScan
      uses: stackhawk/hawkscan-action@v1.2
      with:
        apiKey: ${{ secrets.HAWK_API_KEY }}

push-image:
  needs: [infrastructure-scanning, dast-analysis]
  runs-on: ubuntu-20.04
  steps:
    - uses: actions/checkout@v2
    - name: Log into registry
      run: echo "${{ secrets.GITHUB_TOKEN }}" |
      docker login ghcr.io -u ${{ github.actor }}
      --password-stdin
    - name: Push image to GitHub Registry
      uses: docker/build-push-action@v2
      with:
        context: .
        tags: ghcr.io/jyisus/bestreads-backend:latest
        push: true

deploy-pre:
  needs: push-image
  runs-on: ubuntu-20.04
```

```
steps:
- uses: actions/checkout@v2
- uses: akhileshns/heroku-deploy@v3.12.12
  with:
    heroku_api_key: ${{ secrets.HEROKU_API_KEY }}
    heroku_app_name: bestreads-tfm
    heroku_email: ${{ secrets.HEROKU_EMAIL }}
    usedocker: true
```