Trabajo de Fin de Máster

Máster Universitario en Ciberseguridad e Inteligencia de Datos

DevSecOps - Operaciones de Seguridad en Pipelines de Integración y Entrega Continua

DevSecOps - Security Operations in CI/CD Pipelines

Marvin Correia

La Laguna, 6 de julio de 2024

D. **Cándido Caballero Gil**, profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor.

D.ª **Jezabel Miriam Molina Gil**, profesora Contratado Doctor, adscrita al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutora.

C E R T I F I C A N

Que la presente memoria titulada:

”*DevSecOps - Operaciones de Seguridad en Pipelines de Integración y Entrega Continua*”

ha sido realizada bajo su dirección por D. **Marvin John Dos Santos Correia**, con N.I.E. Z0044179R.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 6 de julio de 2024.

Agradecimientos

En primer lugar, quiero expresar mi más profundo agradecimiento al Gobierno de Canarias, en colaboración con la Fundación Canaria para la Acción Exterior (FUCAEX), por darme la oportunidad de realizar este máster. Su apoyo ha sido fundamental en mi trayectoria académica, y por ello, estoy inmensamente agradecido.

También deseo extender mi más sincero agradecimiento a mi familia por su inquebrantable apoyo y ánimo a lo largo de este esfuerzo. Su amor, paciencia y confianza en mí han sido mi mayor fuente de fortaleza y motivación.

Finalmente, me gustaría reconocer a mis tutores y a todos mis profesores por su guía, conocimiento e inspiración. Su dedicación y compromiso con mi educación han sido invaluables, y estoy profundamente agradecido por el tiempo y esfuerzo que han invertido en mi crecimiento y éxito.

Licencia

Resumen

Incorporar la seguridad en DevOps ha sido un desafío debido a que los métodos de seguridad tradicionales no han podido mantenerse al ritmo de la agilidad y velocidad de DevOps. Para lograrlo, tenía que surgir un nuevo paradigma, con métodos de seguridad modernizados capaces de satisfacer y seguir el mismo ritmo: DevSecOps. Este estudio pretende ofrecer una visión general de lo que es DevSecOps, lo que implica su implementación, los beneficios que se obtienen y los desafíos que enfrentamos al hacerlo. Se analizan las etapas del ciclo de vida del software antes y después de su adopción, destacando los beneficios y desafíos de esta transición. Para ello, realizamos una revisión de los conceptos fundamentales de DevOps y su evolución hacia DevSecOps, subrayando la importancia de integrar la seguridad desde el inicio del desarrollo del software. En el núcleo de la tesis, se diseña y se implementa un pipeline CI/CD robusto en la plataforma Gitlab, que incluye integración continua (CI), entrega continua (CD) y pruebas de seguridad automatizadas, siguiendo la cultura de DevSecOps. Los resultados obtenidos muestran cómo la incorporación de prácticas DevSecOps no sólo mejora la seguridad del software, sino que también optimiza la colaboración entre los equipos de desarrollo, operaciones y seguridad, aplicando un modelo de responsabilidad compartida.

Palabras clave: DevOps, DevSecOps, Ciberseguridad, CI/CD, GitLab

Abstract

Incorporating security into DevOps has been a challenge because traditional security methods have not been able to keep pace with the agility and speed of DevOps. To achieve this, a new paradigm had to emerge, with modernized security methods able to meet and keep pace: DevSecOps. This study aims to provide an overview of what DevSecOps is, what its implementation involves, the benefits to be gained and the challenges we face in performing it. We analyze the stages of the software lifecycle before and after its adoption, highlighting the benefits and challenges of this transition. To do so, we perform a review of the fundamental concepts of DevOps and its evolution towards DevSecOps, highlighting the importance of integrating security from the beginning of software development. At the core of the thesis, a robust CI/CD pipeline is designed and implemented on the Gitlab platform, which includes continuous integration (CI), continuous delivery (CD) and automated security testing, following the DevSecOps culture. The results obtained show how the incorporation of DevSecOps practices not only improves software security, but also optimizes collaboration between development, operations and security teams, applying a shared responsibility model.

Keywords: *DevOps, DevSecOps, Cybersecurity, CI/CD, GitLab*

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

En la era digital actual, la rapidez y la calidad en la entrega de software se han vuelto elementos cruciales para la competitividad de las organizaciones. Las metodologías de desarrollo ágil y las prácticas de Integración y Entrega Continua (CI/CD) han revolucionado la forma en que las aplicaciones son desarrolladas, probadas y desplegadas. Sin embargo, a medida que estas metodologías se adoptan de manera más amplia, surge una necesidad igualmente urgente de integrar la seguridad en cada etapa del ciclo de vida del desarrollo de software.

1.1. Justificativa

La seguridad se ha convertido en un aspecto crítico que no puede ser subestimado. Con la creciente frecuencia y sofisticación de los ciberataques, las organizaciones deben asegurarse de que sus aplicaciones sean seguras desde las primeras etapas de desarrollo hasta su despliegue y operación continua.

Tradicionalmente, las prácticas de seguridad se han abordado al final del ciclo de desarrollo, lo que a menudo resulta en costosos y prolongados procesos de corrección de errores. Esta aproximación no solo incrementa los riesgos de seguridad, sino que también puede afectar la calidad y la eficiencia del desarrollo de software.

La presión regulatoria para garantizar la integridad de todos los componentes de software también está aumentando drásticamente. Las aplicaciones se construyen con un número creciente de componentes de software de código abierto (OSS) y otros artefactos de terceros, cada uno de los cuales puede introducir nuevas vulnerabilidades en la aplicación. Los atacantes buscan explotar las vulnerabilidades de estos componentes, lo que también pone en riesgo a los consumidores del software. [22]

El software representa la mayor superficie de ataque que las organizaciones enfrentan y que no se ha abordado adecuadamente. Algunas estadísticas interesantes para considerar:

- Más del 80 % de las vulnerabilidades de software se introducen a través de software de código abierto (OSS) y componentes de terceros.
- Los ataques a la cadena de suministro digital se están volviendo más agresivos, sofisticados y diversos. Para 2025, el 45 % de las organizaciones habrán experimentado al menos uno (Gartner). [11]
- El costo total de los ciberataques a la cadena de suministro de software para las empresas superará los 80.600 millones de dólares a nivel mundial para 2026, frente a los 45.800 millones en 2023 (Juniper Research).

El entorno de amenazas actual, junto con el impulso para entregar aplicaciones más rápidamente, obliga a las organizaciones a integrar la seguridad en todo el ciclo de vida del desarrollo de software de maneras que no degraden la productividad de los desarrolladores. Esta práctica se conoce formalmente como DevSecOps. [22], [28]

DevSecOps surge como una evolución natural de las metodologías DevOps, integrando la seguridad como una responsabilidad compartida. La filosofía de DevSecOps promueve desplazar la seguridad hacia la izquierda (*Shift security left*), que es una práctica que ayuda a los desarrolladores a encontrar vulnerabilidades y errores de codificación en una fase más temprana del proceso de desarrollo de software[32]. Esta integración es esencial para reducir el tiempo de respuesta ante incidentes de seguridad y asegurar un flujo continuo de entrega de software confiable y seguro.

1.2. Objectivos

1.2.1. Objectivo general

Análisis e implementación de pipelines de CI/CD con integración de capa de seguridad usando las prácticas DevSecOps en la plataforma GitLab.

1.2.2. Objectivos especificos

- Realización de un estudio de caso que demuestre las prácticas DevSecOps en un proyecto de demostración;
- Evaluar el impacto de las operaciones de seguridad en el ciclo de vida del desarrollo;
- *Shifting Security Left*;
- Comparación de las herramientas y tecnologías disponibles para respaldar las operaciones de seguridad en pipelines de CI/CD;
- Proponer pautas y mejores prácticas para implementar DevSecOps;
- Implementar pruebas de seguridad automatizadas.

1.3. Organización de la memoria

En el capítulo 1 (*Introducción* 1), se presentará una visión general del tema en cuestión, incluyendo la evolución del DevOps hacia DevSecOps y la creciente importancia de las operaciones de seguridad en pipelines de CI/CD. También se establecerán los objetivos del estudio y su justificación.

El segundo capítulo (*Fundamentación Teórica* 2) abordará los conceptos fundamentales relacionados con DevSecOps, pipelines de CI/CD y prácticas de seguridad de software. Se discutirán los principales principios y desafíos en la integración de operaciones de seguridad en pipelines de CI/CD, así como las mejores prácticas recomendadas.

En el capítulo 3 (*Estado del Arte* 3), se realizará una revisión detallada de la literatura existente sobre DevSecOps. Se examinarán casos de estudio relevantes de DevSecOps, así como documentos clave en el campo. Además, se explorarán las diferentes plataformas

de CI/CD y las herramientas y tecnologías disponibles para implementar operaciones de seguridad en pipelines de CI/CD, proporcionando una base sólida para el trabajo.

En el capítulo 4 (*Desarrollo e Implementación* 4), se detallará el proceso de desarrollo e implementación de las operaciones de seguridad en el pipeline de CI/CD. Se presentarán las herramientas y tecnologías utilizadas.

En el quinto capítulo (*Resultados y Discusión* 5), se presentarán los resultados obtenidos a partir de la implementación de las operaciones de seguridad en el pipeline de CI/CD. Se discutirán los resultados en relación con los objetivos del estudio.

En el último capítulo (*Conclusiones y líneas futuras* 7), se resumirán los principales resultados y conclusiones del estudio, destacando sus contribuciones al área de DevSecOps. Se ofrecerán recomendaciones para futuras investigaciones y aplicaciones prácticas, cerrando el trabajo de forma concluyente.

Capítulo 2

Fundamento Teórico

En el mundo del desarrollo de software, donde cada línea de código es una oportunidad y cada desafío es una puerta a la innovación, ha surgido un enfoque revolucionario: DevOps, que posteriormente ha evolucionado dando origen a DevSecOps.

2.1. Conceptos de DevOps

DevOps, originado de la unión entre desarrollo (Dev) y operaciones (Ops)[20], nació de la necesidad de superar las barreras tradicionales que separaban estas dos disciplinas en el ciclo de vida del desarrollo de software. Fue impulsado por la búsqueda de mayor agilidad, automatización y colaboración, llevando a un nuevo paradigma donde la entrega continua y la integración continua (CI/CD) se convirtieron en las piedras angulares del proceso de desarrollo.

Imagina un ambiente donde desarrolladores y operadores no son meros colegas de trabajo, sino una sinfonía armoniosa, colaborando para entregar software de calidad de forma rápida y eficiente. Ese es la cultura de DevOps.

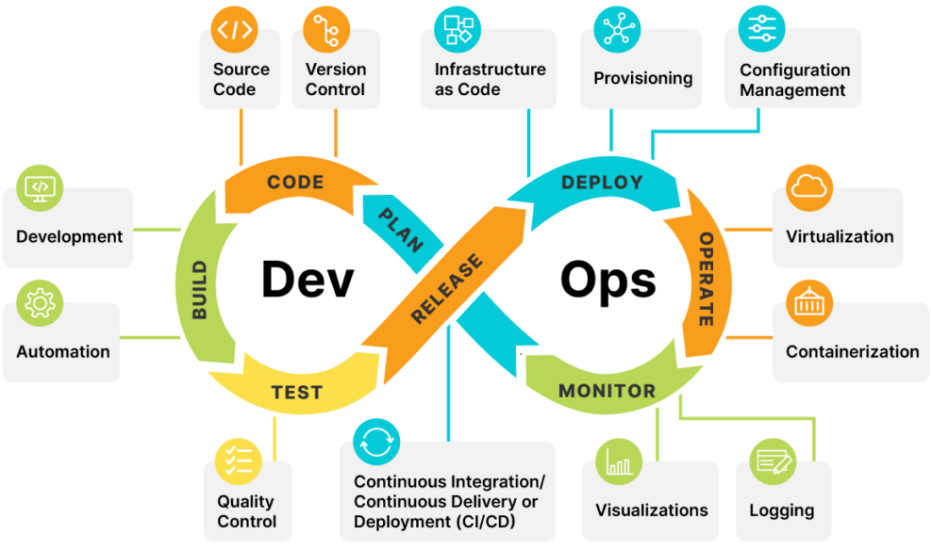

Figura 2.1: Ciclo de vida DevOps (Fuente: Medium [17])

2.1.1. Ciclo de vida del software antes de DevOps

Antes de la implementación de DevOps, el ciclo de vida del software se caracterizaba por una clara división entre los equipos de desarrollo y operaciones. Este enfoque tradicional presentaba varios desafíos:

- **Silos de Información**: Los equipos de desarrollo y operaciones trabajaban de manera aislada, con poca comunicación y colaboración.
- **Procesos Lentos**: La entrega de software era lenta debido a los largos ciclos de desarrollo y las pruebas manuales.
- **Problemas de Integración**: Las integraciones eran muy complejas y propensas a errores, ya que el código desarrollado por diferentes equipos a menudo no se probaba hasta fases posteriores del ciclo de vida.
- **Reacción Tardía a Problemas**: Los problemas y errores se detectaban tarde en el ciclo de vida del desarrollo, lo que hacía que su resolución fuera costosa y lenta.
- **Despliegues Manuales con FTP**: Los despliegues a menudo se realizaban manualmente utilizando herramientas como FTP, lo que era propenso a errores y requería mucho tiempo. Este proceso no tenía visibilidad ni control adecuados, y cualquier cambio realizado podía llevar a inconsistencias y fallos en producción.

2.1.2. Ciclo de vida del software después de DevOps

La adopción de DevOps transformó significativamente el ciclo de vida del software, introduciendo prácticas que mejoraron la eficiencia y la colaboración entre los equipos. Algunos de los cambios clave incluyen:

- **Integración y Entrega Continua (CI/CD)**: Con CI/CD, los cambios en el código se integran, prueban y despliegan de manera continua, lo que reduce el tiempo de entrega y mejora la calidad del software.
- **Automatización de Procesos**: La automatización se implementa en todas las fases del ciclo de vida del desarrollo, desde la construcción y pruebas hasta el despliegue y monitoreo, minimizando los errores humanos y aumentando la eficiencia.
- **Colaboración y Comunicación Mejoradas**: DevOps fomenta una cultura de colaboración donde los desarrolladores y operadores trabajan juntos desde el inicio del proyecto, compartiendo conocimientos y responsabilidades.
- **Feedback Rápido**: Los problemas se detectan y resuelven rápidamente gracias a las pruebas automatizadas y el monitoreo continuo, lo que permite una reacción rápida y mejora continua.
- **Cultura de Responsabilidad Compartida**: Todos los miembros del equipo comparten la responsabilidad de la calidad y la estabilidad del software, lo que lleva a una mayor atención a la seguridad y el rendimiento.
- **Despliegues Automatizados**: En lugar de usar FTP manualmente, el proceso de despliegue se automatiza completamente utilizando herramientas modernas como Docker, Kubernetes a través de pipelines CI/CD. Esto asegura consistencia, control de versiones y un proceso de despliegue mucho más rápido y fiable.

2.2. DevOps y su evolución a DevSecOps

A medida que el mundo digital se expandía y las amenazas cibernéticas se multiplicaban, emergió una nueva dimensión: la seguridad. DevOps, inicialmente enfocado en la velocidad y la entrega continua, reconoció la necesidad imperiosa de integrar la seguridad de forma intrínseca en su ADN. Así nació DevSecOps, una evolución natural que añade una capa crucial de seguridad al proceso de desarrollo de software, como muestra la imagen 2.2.

DevSecOps es un término que se está volviendo cada vez más popular en el mundo del desarrollo de software, y rápidamente se está convirtiendo en la metodología preferida por muchas organizaciones. Es una forma de automatizar el proceso de desarrollo, pruebas, despliegue y mantenimiento de aplicaciones, asegurando al mismo tiempo que se cumplan los requisitos de seguridad y cumplimiento.[35]

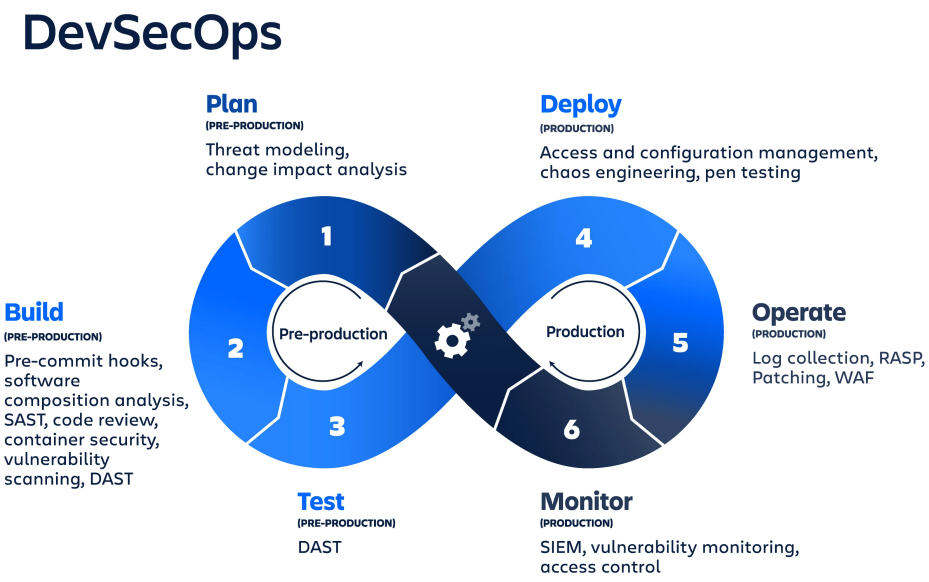

Figura 2.2: Ciclo de vida DevSecOps (Fuente: Medium [35])

DevSecOps no se trata solo de añadir seguridad como una reflexión tardía en el ciclo de vida del desarrollo, sino de incorporarla desde el principio, integrándola perfectamente en cada etapa del proceso.[6]

Es una mentalidad, una cultura, donde la seguridad es vista como responsabilidad de todos y no solo de algunos especialistas aislados[6]. Es la idea de que es posible conciliar velocidad y seguridad, asegurando que el software sea entregado rápidamente sin comprometer su integridad.

2.2.1. Mejores prácticas de DevSecOps

- Automatizar los procesos de seguridad e integrarlos en el pipeline de DevOps.
- Implementar herramientas de escaneo de seguridad para detectar vulnerabilidades.
- Utilizar soluciones de seguridad para asegurar los contenedores.

- Implementar Integración Continua/Entrega Continua (CI/CD).
- Establecer políticas de seguridad y hacerlas cumplir en toda la organización.
- Utilizar análisis de composición de software para identificar componentes de código abierto.
- Adoptar prácticas de codificación segura. [35]

2.3. Pipelines de CI/CD

Los pipelines de CI/CD son una serie de procesos automatizados que permiten la integración continua (CI) y la entrega continua (CD) del software. Estos pipelines son fundamentales en la práctica de DevOps / DevSecOps, ya que garantizan que los cambios en el código se integren, prueben y se desplieguen de manera rápida y eficiente. [13]

2.3.1. Integración Continua (CI)

La integración continua (CI) es una práctica de desarrollo de software en la que los desarrolladores integran regularmente su código en un repositorio compartido. Cada integración se verifica mediante una construcción automatizada y pruebas, lo que permite detectar y solucionar errores rápidamente. [13]

- **Compilación Automática**: Cada vez que un desarrollador envía código al repositorio, se inicia automáticamente un proceso de compilación que asegura que el código se pueda compilar correctamente.
- **Pruebas Automáticas**: Además de la compilación, se ejecutan una serie de pruebas automatizadas para verificar que el nuevo código no introduzca errores o fallos.
- **Feedback Rápido**: La CI proporciona un feedback rápido a los desarrolladores sobre la calidad y funcionalidad de su código, lo que permite detectar y corregir problemas en una etapa temprana.

2.3.2. Entrega Continua (CD)

La entrega continua (CD) extiende los principios de la integración continua para asegurar que el código que ha pasado las pruebas automatizadas se despliegue automáticamente en un entorno de producción o en un entorno de staging listo para la producción.

- **Despliegue Automático**: Una vez que el código ha sido probado y verificado, se despliega automáticamente en el entorno de producción o en un entorno de staging.
- **Monitorización Continua**: Tras el despliegue, se implementan sistemas de monitorización para asegurar que la aplicación funciona correctamente en el entorno de producción y para detectar cualquier problema en tiempo real.

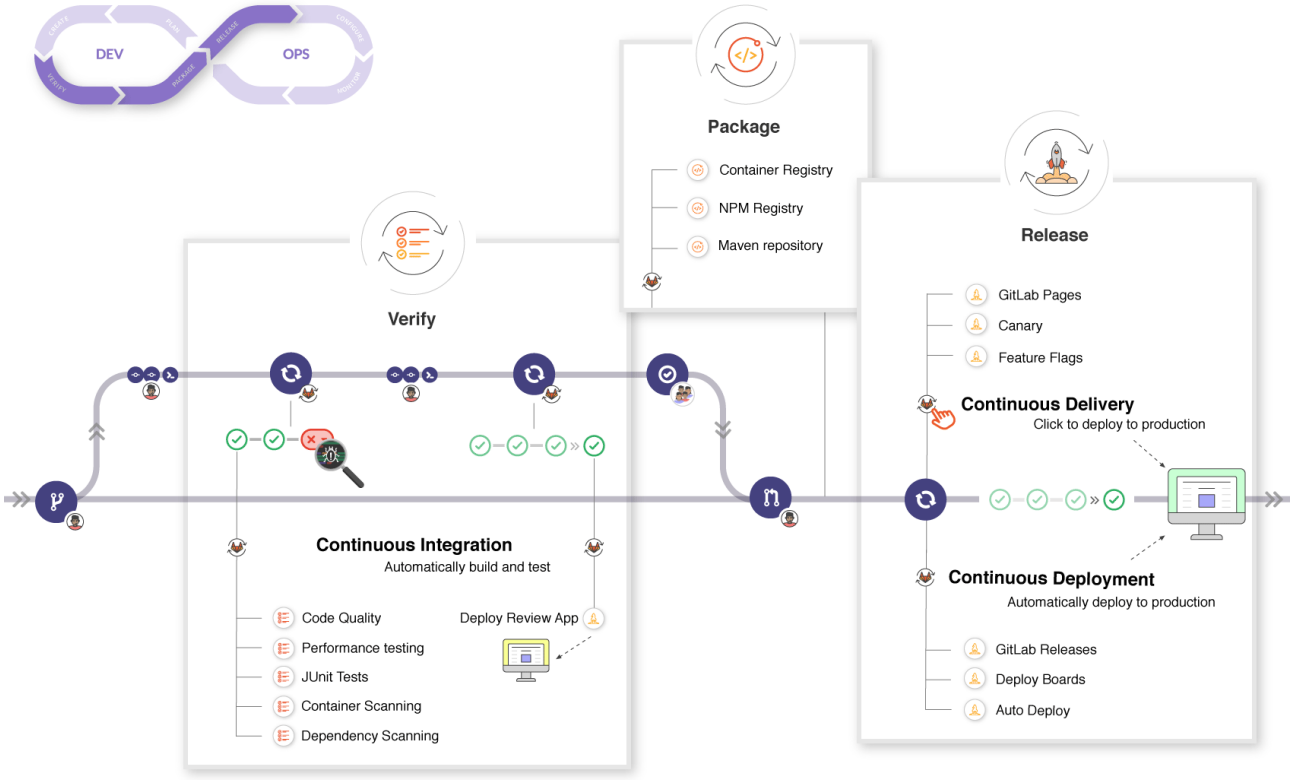

Figura 2.3: Flujo de un Pipeline CI/CD (Fuente: Gitlab [13])

2.3.3. Componentes essenciais de um pipeline de CI/CD

1. **Control de Versión:** El primer paso es tener un sistema robusto de control de versión, como Git, para gestionar el código fuente de forma eficiente. Branches, tags y merges son herramientas esenciales que permiten el desarrollo paralelo y colaborativo.

2. **Automatización de Build:** En esta etapa, se activan scripts de automatización para compilar el código fuente, realizar pruebas unitarias y construir artefactos ejecutables o imagens de contenedores. Herramientas como GitLab CI, Github Action, Azure-Pipelines o Jenkins se utilizan comúnmente para automatizar este proceso.

3. **Pruebas Automatizadas:** Pruebas automatizadas, incluyendo pruebas unitarias, de integración y de seguridad, garantizan la calidad del código y la estabilidad del software. Frameworks como JUnit (Java), unittest (Python), Selenium (Webdriver), Jest (JavaScript), semgrep, sonarqube, son populares para este fin.

4. **Despliegue Automatizado:** Una vez que el código ha sido compilado y probado con éxito, es hora de desplegarlo en entornos de prueba, preproducción y producción. Herramientas de automatización de despliegue, como Ansible, Terraform, Gitlab CI, etc, facilitan el despliegue consistente y fiable en diferentes entornos.

5. **Monitoreo y Registro:** Para garantizar la visibilidad y la transparencia del proceso, es crucial implementar mecanismos de monitoreo y registro. Herramientas como Prometheus, ELK Stack (Elasticsearch, Logstash, Kibana) y Grafana ayudan a

monitorear métricas de rendimiento, identificar problemas y analizar registros de aplicaciones en tiempo real.

6. **Retroalimentación Continua:** Finalmente, el ciclo de CI/CD se completa con retroalimentación continua, proporcionando información valiosa sobre el rendimiento del pipeline, la calidad del código y la experiencia del usuario. Las métricas de rendimiento, los informes de pruebas y la retroalimentación del usuario son fundamentales para orientar mejoras continuas e iteraciones ágiles.

2.4. Desafíos comunes en la implementación de DevSecOps

- **Integración de Seguridad desde el Inicio:** Uno de los desafíos más significativos es integrar la seguridad desde el inicio del ciclo de vida del desarrollo, sin comprometer la velocidad y la eficiencia del pipeline de CI/CD. Muchas veces, la seguridad se ve como una etapa adicional, lo que puede resultar en retrasos en el desarrollo.
- **Identificación de Vulnerabilidades en Tiempo Real:** Identificar y remediar vulnerabilidades de seguridad en tiempo real durante el proceso de desarrollo puede ser un desafío. Es necesario implementar herramientas y prácticas que permitan la detección precoz de amenazas y fallos de seguridad.
- **Conformidad Regulatoria:** Cumplir con los requisitos regulatorios y estándares de seguridad, como GDPR, PCI DSS y HIPAA, puede ser complejo. Garantizar que el pipeline de CI/CD esté en conformidad con estos estándares sin comprometer la agilidad del proceso es un desafío continuo.
- **Automatización de Pruebas de Seguridad:** Automatizar pruebas de seguridad, como pruebas de penetración y análisis estático de código (SAST), puede ser desafiante debido a la complejidad y la diversidad de las aplicaciones y tecnologías involucradas.
- **Gestión de Claves y Credenciales:** Garantizar una gestión segura de claves de API, credenciales de acceso y otros secretos durante el pipeline de CI/CD es fundamental para evitar filtraciones de datos y violaciones de seguridad.
- **Control de Acceso y Autorización:** Gestionar el acceso a los recursos del pipeline de CI/CD y garantizar que solo usuarios autorizados puedan realizar cambios o ejecutar tareas críticas es un desafío constante en términos de seguridad.
- **Monitoreo y Análisis de Seguridad:** Implementar mecanismos de monitoreo y análisis de seguridad en tiempo real para detectar actividades sospechosas y responder a incidentes de seguridad de manera eficaz es crucial, pero puede ser complejo de configurar y mantener.
- **Educación y Concienciación del Equipo:** Educar y concienciar al equipo de desarrollo sobre las mejores prácticas de seguridad y los riesgos asociados es un desafío continuo, pero esencial para garantizar una cultura de seguridad fuerte en toda la organización.

2.5. Conceptos Fundamentales de Ciberseguridad

En esta sección, definiremos algunos de los conceptos fundamentales de ciberseguridad que son relevantes para la implementación de DevSecOps en pipelines de CI/CD. Estos conceptos abarcan una variedad de aspectos relacionados con la identificación, evaluación y mitigación de vulnerabilidades de seguridad.

2.5.1. Vulnerabilidades

Las vulnerabilidades son debilidades o fallos en el software que pueden ser explotados por atacantes para comprometer la integridad, confidencialidad o disponibilidad de los sistemas. Las vulnerabilidades pueden surgir debido a errores en el código, configuraciones incorrectas o falta de actualizaciones.[25]

Ejemplos de vulnerabilidades:

- Desbordamiento del búfer (Buffer Overflow);
- Validación de datos inadecuada;
- Deserialización de datos no fiables;
- Transporte inseguro (páginas no protegidas por SSL);
- Tratamiento de errores inadecuada;

2.5.2. Ataques de Ciberseguridad

Los ataques son las técnicas que los atacantes utilizan para explotar las vulnerabilidades en las aplicaciones. A menudo, los ataques se confunden con las vulnerabilidades, por lo que se debe asegurar de que el ataque descrito sea algo que un atacante haría, en lugar de una debilidad en una aplicación. [24]

2.5.3. Tipos de Ataques

Existen varios tipos de ataques que los atacantes pueden llevar a cabo para explotar vulnerabilidades:

- **Inyección SQL:** Un tipo de ataque en el que un atacante inserta código SQL malicioso en una consulta para acceder o manipular la base de datos.
- **Cross-Site Scripting (XSS):** Un ataque que permite a un atacante inyectar scripts maliciosos en una página web.
- **Phishing:** Un método en el que los atacantes engañan a los usuarios para que revelen información sensible, como contraseñas o datos bancarios.
- **Clickjacking:** Un ataque en el que los atacantes engañan a los usuarios para que hagan clic en elementos de una página web diferentes a los que pretendían, permitiendo a los atacantes ejecutar acciones no autorizadas.
- **Ataques de Denegación de Servicio (DoS):** Ataques que buscan hacer que un servicio o red no esté disponible para sus usuarios legítimos.

2.5.4. Pruebas de Seguridad

Las pruebas de seguridad son esenciales para identificar y mitigar vulnerabilidades antes de que sean explotadas por atacantes. A continuación, se describen tres de los distintos tipos de pruebas de seguridad:

Pruebas de Seguridad Estáticas (SAST)

Las pruebas SAST analizan el código fuente de una aplicación para identificar vulnerabilidades de seguridad sin ejecutarlo. Estas pruebas permiten detectar problemas como inyecciones de SQL, XSS y malas prácticas de codificación desde las primeras etapas del desarrollo.

Análisis de Composición de Software (SCA)

El SCA se centra en analizar las dependencias y componentes de terceros utilizados en una aplicación. Este análisis ayuda a identificar vulnerabilidades conocidas en librerías y frameworks que forman parte del software.

Pruebas de Seguridad Dinámicas (DAST)

Las pruebas DAST se llevan a cabo en aplicaciones en ejecución para identificar vulnerabilidades que solo se pueden detectar durante la ejecución, como fallos de configuración y problemas en la lógica de negocio.

2.5.5. Diferencias entre CVSS, CVE y CWE

[18]

- **CVSS (Common Vulnerability Scoring System):** Es un estándar para evaluar la severidad de las vulnerabilidades de seguridad. Proporciona una puntuación que ayuda a priorizar las vulnerabilidades en función de su impacto.
- **CVE (Common Vulnerabilities and Exposures):** Es una lista de identificadores únicos para vulnerabilidades de seguridad conocidas. Cada CVE proporciona detalles sobre una vulnerabilidad específica, incluidos los productos afectados y las posibles mitigaciones.
- **CWE (Common Weakness Enumeration):** Es una categorización de tipos de fallos de software que pueden conducir a vulnerabilidades. Los CWEs ayudan a los desarrolladores a entender y mitigar debilidades específicas en el diseño y la implementación del software.

Capítulo 3

Estado del arte

El campo de DevSecOps ha evolucionado significativamente en los últimos años, impulsado por la necesidad de integrar prácticas de seguridad dentro del ciclo de vida del desarrollo de software. Esta sección ofrece una revisión de la literatura relevante sobre DevSecOps, analiza casos de estudio y trabajos de investigación relacionados, y examina las herramientas y tecnologías disponibles para implementar operaciones de seguridad en pipelines de CI/CD.

Los estudios de caso son fundamentales para entender cómo las organizaciones han adoptado DevSecOps. Empresas como AWS, Microsoft y Gitlab han compartido sus experiencias y lecciones aprendidas al implementar estas prácticas.

3.1. Casos de Estudio Relevantes de DevSecOps

En esta sección, revisamos algunos casos de estudio relevantes de grandes empresas que han implementado prácticas DevSecOps exitosamente, así como documentos y contribuciones al campo.

3.1.1. Amazon Web Service (AWS)

Amazon ha incorporado prácticas de DevSecOps en sus servicios de AWS, promoviendo la automatización de la seguridad y el cumplimiento mediante herramientas como AWS CodePipeline y AWS Security Hub.

- Caso de Estudio: Integración de DevSecOps en AWS
- Referencia: *Building end-to-end AWS DevSecOps CI/CD pipeline with open source SCA, SAST and DAST tools* [5]

3.1.2. Microsoft

Microsoft ha adoptado DevSecOps en sus plataformas Azure DevOps y Github, proporcionando herramientas de análisis de seguridad y cumplimiento como Azure Security Center, Azure DevOps Pipelines y github actions.

- Caso de Estudio: DevSecOps with Azure and GitHub
- Referencia: *Enable DevSecOps with Azure and GitHub* [8]

3.1.3. Gitlab

GitLab permite a sus equipos equilibrar velocidad y seguridad mediante la automatización de la entrega de software y la protección de toda la cadena de suministro de software.

- Caso de Estudio: *Shift Left Security and Compliance*
- Referencia: *Shift Left Security and Compliance* [14]

3.1.4. Documentos Clave

A continuación, enumeramos algunos documentos y artículos que han contribuido significativamente al campo de DevSecOps:

- Kim, Gene, et al. *The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations*. IT Revolution Press, 2016. [20]
- Forsgren, Nicole, et al. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. IT Revolution Press, 2018. [10]
- Bird, Jim. *DevSecOps: A leader's guide to producing secure software without compromising flow, feedback and continuous improvement*. O'Reilly Media, 2019. [6]
- Shortridge, Kelly, et al. *Security Chaos Engineering: Sustaining Resilience in Software and Systems*. O'Reilly Media, 2020. [30]

3.2. Plataformas de CI/CD para Implementar DevSecOps

En el panorama siempre cambiante del desarrollo de software, elegir la plataforma adecuada puede ser todo un reto. En esta sección vamos examinar algunas de las mejores plataformas de CI/CD que facilitan las prácticas de DevSecOps, centrándose en sus características, ventajas y cómo contribuyen al desarrollo de software seguro. [2], [4], [9], [34]. En la Figura 3.1 se enumeran las 20 herramientas de CI/CD más populares.

3.2.1. GitHub Actions

GitHub Actions ofrece una poderosa plataforma de automatización integrada directamente con los repositorios de GitHub. Permite a los desarrolladores crear flujos de trabajo personalizados para construir, probar y desplegar código, lo que lo convierte en una herramienta robusta para CI/CD y DevSecOps.

Características:

- *Integración Nativa:* Se integra perfectamente con GitHub, proporcionando un entorno cohesivo para los flujos de trabajo de desarrollo y seguridad.

- *Pruebas de Seguridad:* Soporta numerosas herramientas de seguridad como Snyk, Dependabot y CodeQL para identificar vulnerabilidades desde el inicio del ciclo de desarrollo.
- *Extensibilidad:* Ofrece un marketplace con cientos de acciones preconstruidas, incluidas las de escaneo de seguridad y verificación de cumplimiento.

Figura 3.1: Top 20 Herramientas CI/CD (Fuente: mindbowser [21])

Ventajas:

- Simplifica la incorporación de controles de seguridad dentro del pipeline CI/CD.
- Reduce la complejidad de configuración al aprovechar la infraestructura existente de GitHub.
- Facilita la colaboración entre equipos de desarrollo y seguridad mediante el seguimiento integrado de problemas y revisiones de código.

3.2.2. Azure Pipelines

Azure Pipelines, parte del conjunto de herramientas de Azure DevOps, soporta la construcción, prueba y despliegue de código en cualquier plataforma. Ofrece un amplio soporte para integrar herramientas de seguridad, lo que lo convierte en una fuerte opción para DevSecOps.

Características:

- *Soporte Multiplataforma:* Compatible con diversos lenguajes de programación y plataformas.
- *Herramientas de Seguridad Integradas:* Incluye integraciones con escáneres de seguridad como WhiteSource Bolt y otras herramientas de análisis estático.
- *Cumplimiento y Gobernanza:* Proporciona características para asegurar el cumplimiento con estándares de la industria a través de verificaciones automáticas y la aplicación de políticas.

Ventajas:

- Flexible y escalable, capaz de manejar pipelines de despliegue complejos.
- Mejora la seguridad con pruebas integradas y herramientas de cumplimiento.
- Soporta trazabilidad y auditoría de extremo a extremo para una mejor gobernanza.

3.2.3. GitLab CI/CD

GitLab CI/CD es parte de la plataforma todo-en-uno de DevOps de GitLab, que ofrece herramientas robustas para CI/CD, gestión de código fuente y pruebas de seguridad. Su completo conjunto de características soporta todo el ciclo de vida de DevSecOps.

Características:

- *Seguridad Integrada:* Incluye herramientas SAST, DAST y de escaneo de contenedores directamente en la plataforma.
- *Automatización:* Automatiza las pruebas de seguridad en cada etapa del pipeline CI/CD.
- *Colaboración:* Facilita la colaboración sin problemas a través de solicitudes de merge, revisiones de código y seguimiento de problemas integrados.

Ventajas:

- Consolida todas las herramientas de DevSecOps en una sola plataforma, reduciendo la necesidad de múltiples integraciones.
- Automatiza las pruebas de seguridad, ayudando a identificar vulnerabilidades de manera temprana y frecuente.

- Mejora la productividad del desarrollador con una interfaz unificada y flujos de trabajo optimizados.

3.2.4. Jenkins

Jenkins es un servidor de automatización de código abierto ampliamente utilizado que soporta la construcción, prueba y despliegue de software. Con su extenso ecosistema de plugins, Jenkins puede ser personalizado para soportar prácticas de DevSecOps de manera efectiva.

Características:

- *Ecosistema de Plugins:* Más de 1,000 plugins disponibles, incluidos los de escaneo de seguridad y cumplimiento.
- *Personalización:* Altamente flexible y personalizable para adaptarse a diversas necesidades de CI/CD.
- *Soporte Comunitario:* Gran respaldo de la comunidad y amplia documentación.

Ventajas:

- Altamente adaptable a requisitos y entornos específicos del proyecto.
- Ofrece capacidades robustas de automatización, reduciendo la intervención manual.
- Permite integraciones completas de seguridad a través de plugins, soportando una amplia gama de herramientas de seguridad.

3.2.5. CircleCI

CircleCI proporciona una plataforma CI/CD basada en la nube conocida por su velocidad y escalabilidad. Soporta flujos de trabajo automatizados de prueba y despliegue, lo que lo hace adecuado para integrar prácticas de seguridad.

Características:

- *Rendimiento:* Optimizado para construcciones y despliegues rápidos.
- *Integraciones de Seguridad:* Soporta herramientas como Snyk, Aqua Security y otras para verificaciones continuas de seguridad.
- *Facilidad de Uso:* Interfaz y proceso de configuración amigables, facilitando la adopción rápida.

Ventajas:

- Acelera los ciclos de desarrollo con procesos de construcción eficientes.
- Garantiza seguridad continua a través de la integración sin problemas de herramientas de seguridad.

- Simplifica la configuración de pipelines con archivos de configuración YAML reutilizables.

3.2.6. Bitbucket Pipelines

Bitbucket Pipelines es una plataforma de CI/CD integrada en Bitbucket, diseñada para facilitar la entrega continua de software. Al estar integrada con Bitbucket, proporciona un entorno cohesivo para el desarrollo y la implementación.

Características:

- *Integración con Bitbucket:* Se integra perfectamente con los repositorios de Bitbucket, ofreciendo una experiencia fluida de desarrollo a despliegue.
- *Configuración Sencilla:* Utiliza archivos YAML para definir las pipelines, lo que simplifica la configuración y la gestión.
- *Herramientas de Seguridad:* Soporta integraciones con herramientas de seguridad como Snyk para el análisis de vulnerabilidades.

Ventajas:

- Facilita la integración de seguridad en el proceso de desarrollo continuo.
- Simplifica la configuración y el manejo de pipelines con una sintaxis YAML clara y concisa.
- Permite una colaboración eficiente entre los equipos de desarrollo y operaciones gracias a la integración con Bitbucket.

Implementar DevSecOps requiere elegir la plataforma de CI/CD adecuada que no solo soporte el desarrollo eficiente de software sino que también integre prácticas de seguridad robustas.

3.3. Herramientas y tecnologías para implementar operaciones de seguridad en pipelines de CI/CD

Hay una amplia gama de herramientas y tecnologías diseñadas para integrar la seguridad en los pipelines de CI/CD. Estas herramientas ayudan a automatizar pruebas de seguridad, analizar vulnerabilidades y garantizar el cumplimiento de políticas de seguridad.

La implementación de DevSecOps en pipelines de CI/CD requiere el uso de diversas herramientas y tecnologías. A continuación, se describen algunas de las mejores y más utilizadas herramientas en 2024:

3.3.1. SonarQube

SonarQube es una plataforma de análisis estático de código que permite detectar errores de programación, vulnerabilidades de seguridad y problemas de calidad del

código. [33]
Características:

- Análisis de código en tiempo real.
- Soporte para múltiples lenguajes de programación.
- Informes detallados y paneles de control.
- Detecta vulnerabilidades de seguridad temprano en el ciclo de desarrollo.
- Fácil integración con sistemas de CI/CD populares como Jenkins, GitLab CI/CD y Azure DevOps.

En la figura 3.2 se enumeran las 11 herramientas de pruebas de sast más comunes en 2024.

Figura 3.2: Top 11 Herriamientas SAST (Fuente: expertinsights[19])

3.3.2. GitLab CI/CD

GitLab CI/CD es una plataforma completa de DevOps que permite la integración continua y la entrega continua. Ofrece herramientas de gestión de código fuente, integración de pipelines y seguridad integrada. [16]
Características:

- Gestión de repositorios de código.
- Pipelines de CI/CD configurables.
- Escaneo de seguridad y análisis de composición de software.
- Monitoreo y métricas de rendimiento.

- Plataforma todo en uno para DevOps.
- Seguridad integrada a través de análisis SAST, DAST y SCA.
- Amplia comunidad de soporte y documentación.

3.3.3. OWASP ZAP (Zed Attack Proxy)

OWASP ZAP es una herramienta de pruebas de seguridad para aplicaciones web que ayuda a encontrar vulnerabilidades en aplicaciones web durante el desarrollo y las pruebas. [26]

Características:

- Escaneo de aplicaciones web en busca de vulnerabilidades.
- Integración con pipelines de CI/CD.
- Herramientas de análisis automatizado y manual.
- Informes detallados de seguridad.
- Fuente abierta y gratuita.
- Potente conjunto de herramientas para pruebas de penetración.
- Fácil de integrar con herramientas de CI/CD como Jenkins y GitLab.

3.3.4. Snyk

Snyk es una herramienta de seguridad que se enfoca en identificar y corregir vulnerabilidades en dependencias de código abierto. [31]

Características:

- Escaneo de vulnerabilidades en dependencias de código abierto.
- Monitoreo continuo de vulnerabilidades.
- Generación de parches automáticos para vulnerabilidades.
- Integración con repositorios de código y herramientas de CI/CD.
- Proporciona soluciones rápidas para vulnerabilidades conocidas.
- Integración sin interrupciones con herramientas de desarrollo y CI/CD.
- Mantiene actualizadas las dependencias de código abierto.

3.3.5. Aqua Security

Aqua Security proporciona soluciones de seguridad para aplicaciones basadas en contenedores, asegurando que los contenedores y las funciones sin servidor sean seguras a lo largo de todo el ciclo de vida. [3]
Características:

- Escaneo de imágenes de contenedores en busca de vulnerabilidades.
- Monitoreo de tiempo de ejecución para detectar comportamientos anómalos.
- Políticas de cumplimiento y gestión de secretos.
- Integración con herramientas de orquestación de contenedores como Kubernetes.
- Solución integral para la seguridad de contenedores.
- Identificación y mitigación proactiva de amenazas.
- Integración con los principales proveedores de nube y plataformas de orquestación.

3.3.6. Checkmarx

Checkmarx es una plataforma de seguridad de aplicaciones que ofrece herramientas para realizar pruebas de seguridad estáticas (SAST) y dinámicas (DAST), así como análisis de composición de software (SCA). [7]
Características:

- Análisis estático de código para detectar vulnerabilidades.
- Escaneo dinámico de aplicaciones en tiempo real.
- Análisis de la composición del software para identificar componentes de código abierto vulnerables.
- Integración con sistemas de CI/CD y repositorios de código.
- Amplia cobertura de seguridad para diferentes tipos de aplicaciones.
- Integración fácil con herramientas de CI/CD.
- Soporte para múltiples lenguajes de programación.

3.3.7. Aikido

Aikido es una herramienta de automatización de seguridad que permite la integración de prácticas de seguridad en pipelines de CI/CD mediante la automatización de pruebas y auditorías de seguridad. [1]
Características:

- Automatización de pruebas de seguridad.
- Auditorías de seguridad continuas.
- Integración con herramientas de CI/CD y repositorios de código.

- Informes detallados de seguridad y cumplimiento.
- Facilita la integración continua de la seguridad en el ciclo de desarrollo.
- Reducción del tiempo de respuesta ante vulnerabilidades.
- Mejora del cumplimiento de normativas y estándares de seguridad.

3.3.8. Veracode

Veracode es una plataforma de seguridad de aplicaciones que ofrece servicios de análisis estático y dinámico de seguridad, así como análisis de composición de software y pruebas de penetración. [36]
Características:

- Análisis estático y dinámico de seguridad.
- Pruebas de penetración automatizadas.
- Análisis de composición de software para identificar vulnerabilidades en componentes de código abierto.
- Informes de seguridad y métricas de cumplimiento.
- Amplia gama de servicios de seguridad de aplicaciones.
- Integración con herramientas de CI/CD y repositorios de código.
- Mejora continua de la seguridad del software mediante pruebas automatizadas.

La siguiente tabla muestra una lista de herramientas categorizadas en tres tipos principales de pruebas de seguridad: SAST (Static Application Security Testing), SCA (Software Composition Analysis) y DAST (Dynamic Application Security Testing).

SAST	**SCA**	**DAST**
SonarQube	Snyk	OWASP ZAP
Fortify	Black Duck	Burp Suite
Checkmarx	Dependabot	Acunetix
CodeQL	Whitesource	Nikto
Bandit	Trivy	Netsparker
Semgrep	OWASP Dependency-Check	W3af
Veracode	FOSSA	AppSpider
ShiftLeft	JFrog Xray	Arachni
Brakeman	Sonatype Nexus	Qualys
Aikido	SCA Maven Plugin	WebInspect

Tabla 3.1: Tecnologías por tipos de pruebas de seguridad

Capítulo 4

Desarrollo e Implementación

En este capítulo, desarrollaremos un caso de uso, utilizando un proyecto vulnerable llamado PyGoat [27], desarrollado en Python utilizando el framework Django.

Emplearemos GitLab como plataforma DevOps y aprovecharemos sus capacidades para implementar pipelines de seguridad a través de GitLab CI/CD. El objetivo es construir, detectar vulnerabilidades y realizar despliegues continuos, integrando prácticas DevSecOps en el proceso.

4.1. Etapas para Creación de un Pipeline CI/CD con Prácticas DevSecOps en GitLab

El pipeline será responsable de construir el proyecto, hacer pruebas de seguridad para detectar vulnerabilidades y desplegarlo de manera continua. Nos enfocaremos en la integración de herramientas y pruebas de seguridad que permitan asegurar el código desde las primeras etapas del desarrollo.

4.1.1. Etapas del Pipeline

Nuestro pipeline se estructurará en varias etapas clave, cada una con un propósito específico, en gitlab cada etapa es llamado de *stage*:

- **Build**: Construcción del proyecto para asegurar que todos los componentes se ensamblan correctamente.
- **Static Security Test (SAST/ SCA)**: Realización de pruebas de seguridad estática y análisis de la composición del software para identificar vulnerabilidades en el código y en las dependencias.
- **Deploy**: Despliegue continuo del proyecto en el entorno de producción o de pruebas.
- **Dynamic Application Security Testing (DAST)**: Pruebas de seguridad dinámica para identificar vulnerabilidades en la aplicación en funcionamiento.

4.1.2. Pasos para Implementar el Pipeline DevSecOps

Para implementar este pipeline DevSecOps, seguiremos una serie de pasos bien definidos:

1. **Crear y Configurar el Repositorio en GitLab**: Establecer el repositorio del proyecto PyGoat en GitLab.com y configurar los ajustes iniciales necesarios.
2. **Build**: Construcción del proyecto usando la herriamienta Docker.
3. **SAST (Static Application Security Testing)**: Análisis estático del código fuente para identificar vulnerabilidades.
4. **SCA (Software Composition Analysis)**: Análisis de la composición del software para detectar vulnerabilidades en las dependencias.
5. **Scaneo de Secretos**: Detección de posibles filtraciones de secretos y credenciales en el código fuente.
6. **Deploy**: Despliegue continuo del proyecto en el entorno designado.
7. **DAST (Dynamic Application Security Testing)**: Pruebas de seguridad dinámica para identificar vulnerabilidades en la aplicación en funcionamiento.

4.2. Crear y Configurar el Repositorio

El primer paso será crear y configurar el repositorio de código del proyecto PyGoat en *gitLab.com*. A partir de ahí, procederemos a construir un pipeline CI/CD que incorpore prácticas DevSecOps.

Para importar el proyecto PyGoat [27] desde el repositorio original en GitHub a GitLab utilizando la funcionalidad de importación de proyectos vía URL, como muestra la imagen 4.1.

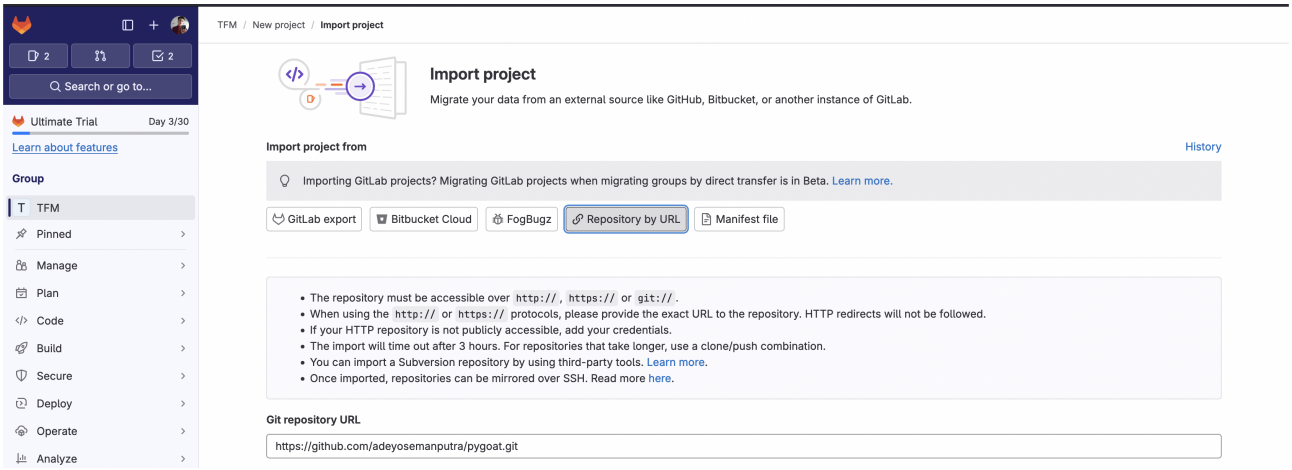

Figura 4.1: Importación del proyecto PyGoat (Fuente: capt. de pantalla Gitlab)

Para configurar CI/CD en un repositorio utilizando GitLab, es necesario crear un archivo `.gitlab-ci.yml` que contenga las definiciones de los stages (etapas) y jobs (trabajos) para el pipeline. Aquí está un ejemplo de cómo estructurar este archivo para incluir los stages build, test, deploy, y dast:

```
1 # Archivo .gitlab-ci.yml
2
3 # Definición de los stages (etapas) del pipeline
```

```yaml
stages:
  - build
  - test
  - deploy
  - dast

# Definición de los jobs (trabajos) para cada stage
build_job:
  stage: build
  script:
    - echo "Building the project..."

test_job:
  stage: test
  script:
    - echo "Running tests..."

deploy_job:
  stage: deploy
  script:
    - echo "Deploying the application..."

dast_job:
  stage: dast
  script:
    - echo "Running Dynamic Application Security Testing (DAST)..."
```

El código *yaml* resultaría en un pipeline con varias etapas y trabajos como se muestra en la figura 4.2:

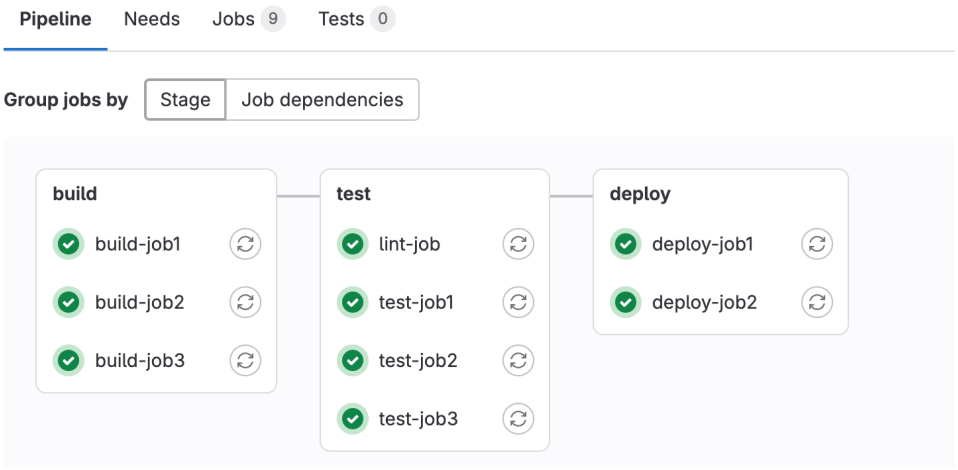

Figura 4.2: Ejemplo de Jobs de CICD (Fuente: Gitlab Pipelines [14])

Este archivo `.gitlab-ci.yml` actúa como la configuración central para automatizar y gestionar el ciclo de vida del desarrollo del software mediante CI/CD en GitLab. Cada vez que se haga un push a este repositorio, GitLab ejecutará automáticamente los jobs definidos en cada stage según la configuración especificada.

4.3. Build: Creación de la Imagen Docker

En esta etapa, el objetivo es crear una imagen Docker que empaquete el proyecto. Para ello, se debe crear un archivo `Dockerfile` que contenga las instrucciones necesarias para construir la imagen. Además, se configurará un job de build en el archivo `.gitlab-ci.yml` para construir y subir la imagen al GitLab Container Registry.

4.3.1. Creación del Dockerfile

Un `Dockerfile` es un archivo de texto que contiene una serie de instrucciones para crear una imagen Docker. A continuación mostramos el Dockerfile para empaquetar la aplicación web PyGoat:

```
FROM python:3.11-slim-buster
WORKDIR /app
# Install dependencies
RUN pip install pipenv
COPY Pipfile Pipfile.lock ./
RUN pipenv install --system --deploy
# copy source code
COPY . .
# define app run command
EXPOSE 8000
CMD python manage.py migrate && gunicorn --bind 0.0.0.0:8000 pygoat.wsgi
```

4.3.2. Configuración del Job de Build en .gitlab-ci.yml

Para construir la imagen Docker y subirla al GitLab Container Registry, se debe agregar un job de build en el archivo `.gitlab-ci.yml`:

```
# Archivo .gitlab-ci.yml
# Definición de los stages (etapas) del pipeline
stages:
  - build
  - test
  - deploy
  - dast

# Job para construir la imagen Docker
docker-build:
  stage: build
  image: docker:cli
  stage: build
  services:
    - docker:dind
  variables:
    DOCKER_IMAGE_NAME: $CI_REGISTRY_IMAGE:$CI_COMMIT_REF_SLUG
  before_script:
```

```
    - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD"
      ↪  $CI_REGISTRY
  script:
    - docker build --pull -t "$DOCKER_IMAGE_NAME" .
    - docker push "$DOCKER_IMAGE_NAME"
  # Run this job in a branch where a Dockerfile exists
  rules:
    - if: $CI_COMMIT_BRANCH
      exists:
        - Dockerfile
  needs: []
```

El build job realiza las siguientes tareas:

- **Preparativos (Before Script)**: Antes de empezar, el sistema inicia sesión en el almacén de imágenes docker (Gitlab Registry), asegurándose de que tiene permiso para subir y acceder a las imágenes. Esta acción es similar a ingresar a una cuenta en una plataforma en línea antes de poder guardar o acceder a los archivos.
- **Construcción y Subida (Script)**: El sistema procede a empaquetar la aplicación en una imagen Docker, seguindo las instrucciones nel fichero `Dockerfile` .Este proceso se puede comparar con empaquetar todos los archivos y configuraciones necesarios de un proyecto en una caja, lo que permite que el proyecto sea utilizado o movido fácilmente. Tras la construcción, la imagen se guarda en el repositorio de imágenes para poder acceder a ella más tarde.

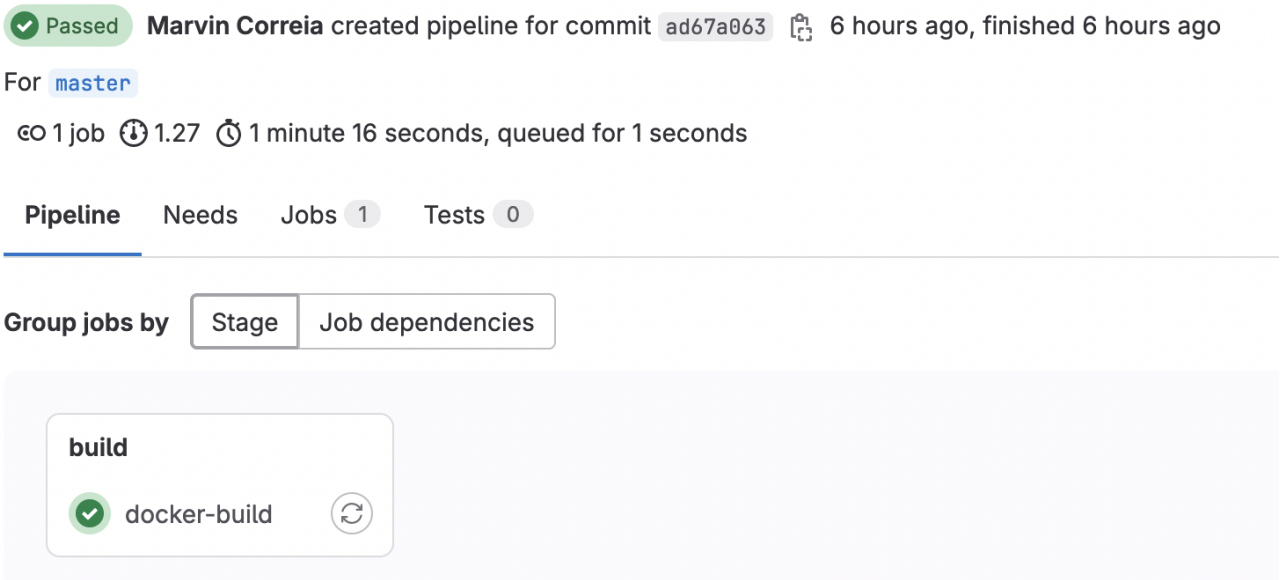

Figura 4.3: Build Job (Fuente: capt. de pantalla Gitlab)

4.4. Pruebas Estáticas de Seguridad

En esta sección, abordaremos la etapa de pruebas de seguridad en nuestro pipeline CI/CD, incluyendo SAST, SCA y detección de secretos.

4.4.1. SAST (Static Application Security Testing)

SAST es un método de análisis de seguridad que examina el código fuente de la aplicación sin ejecutarlo. Este tipo de prueba se realiza para identificar vulnerabilidades de seguridad en las primeras etapas del desarrollo, permitiendo a los desarrolladores identificar y corregir errores antes de que el software sea desplegado.

Templates de CI/CD en GitLab

GitLab facilita la configuración de pipelines de CI/CD ofreciendo un conjunto de plantillas predefinidas. Estas plantillas contienen configuraciones y jobs comunes que pueden ser fácilmente importados en nuestro fichero de configuracción `.gitlab-ci.yml`, utilizando la directiva `include`. Para SAST, GitLab proporciona una plantilla por defecto que podemos integrar en nuestro pipeline. [12]

Integración de la Plantilla SAST en GitLab

Para utilizar la plantilla SAST por defecto en GitLab, añadiremos la siguiente configuración a nuestro archivo `.gitlab-ci.yml`:

```yaml
# Archivo .gitlab-ci.yml
stages:
  - build
  - test
  ...
# Include SAST JOBS
include:
  - template: Security/SAST.gitlab-ci.yml

# ... other jobs
```

Figura 4.4: Código fuente del template de SAST (Fuente: Gitlab Templates[12])

Esta plantilla automatiza el proceso de análisis estático de seguridad, ejecutando un conjunto de herramientas configuradas por GitLab para identificar vulnerabilidades en el código fuente.

El job `sast` en el pipeline se encarga de realizar pruebas estáticas de seguridad de aplicaciones en el código fuente del proyecto. A continuación, se detalla la configuración de este job y su funcionamiento:

- **Etapa de Ejecución**: El job `sast` se ejecuta en la etapa de pruebas (`test`) del pipeline CI/CD.
- **Generación de Artefactos**: Durante la ejecución del job, se genera un archivo de informe SAST llamado `gl-sast-report.json`.
- **Acceso a Artefactos**: Los artefactos generados están accesibles para los desarrolladores, permitiéndoles revisar y analizar los resultados del análisis de seguridad.
- **Informe SAST**: El informe `gl-sast-report.json` proporciona detalles sobre las vulnerabilidades de seguridad encontradas en el código fuente, permitiendo a los desarrolladores tomar medidas correctivas.

Semgrep

Una de las herramientas utilizadas por la plantilla SAST de GitLab es Semgrep [29]. Semgrep es una herramienta de análisis estático de código que permite buscar patrones específicos en el código fuente. Es muy eficaz para identificar problemas de seguridad, buenas prácticas y errores comunes en varios lenguajes de programación.

Creación de un Job SAST Específico con SonarQube

Además de utilizar la plantilla SAST por defecto de GitLab, vamos a crear un job específico para realizar un análisis estático de código con SonarQube. SonarQube es una plataforma de código abierto que proporciona revisiones continuas de calidad de código y detección automática de bugs, vulnerabilidades y *code smells*.

A continuación, se muestra cómo integrar SonarQube en nuestro pipeline CI/CD:

```yaml
# Archivo .gitlab-ci.yml
# ...
sonarqube-quality-gate:
  stage: test
  image:
    name: sonarsource/sonar-scanner-cli:5.0
    entrypoint: [""]
  variables:
    SONAR_USER_HOME: "${CI_PROJECT_DIR}/.sonar"
    GIT_DEPTH: "0"
  cache:
    key: "${CI_JOB_NAME}"
    paths:
      - .sonar/cache
  script:
```

```
    - sonar-scanner
  allow_failure: true
  needs:
    - []
  only:
    - merge_requests
    - master
    - main
    - develop
```

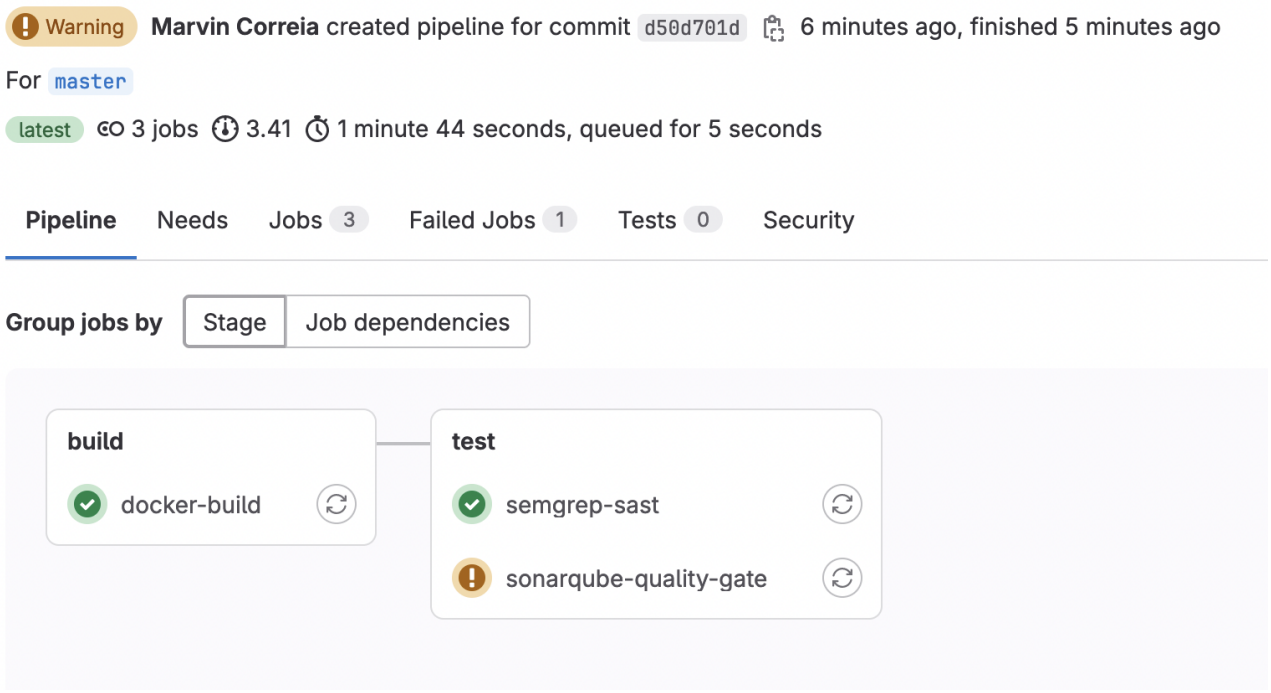

Figura 4.5: Job de SAST con Semgrep y SonarQube (Fuente: capt. de pantalla Gitlab)

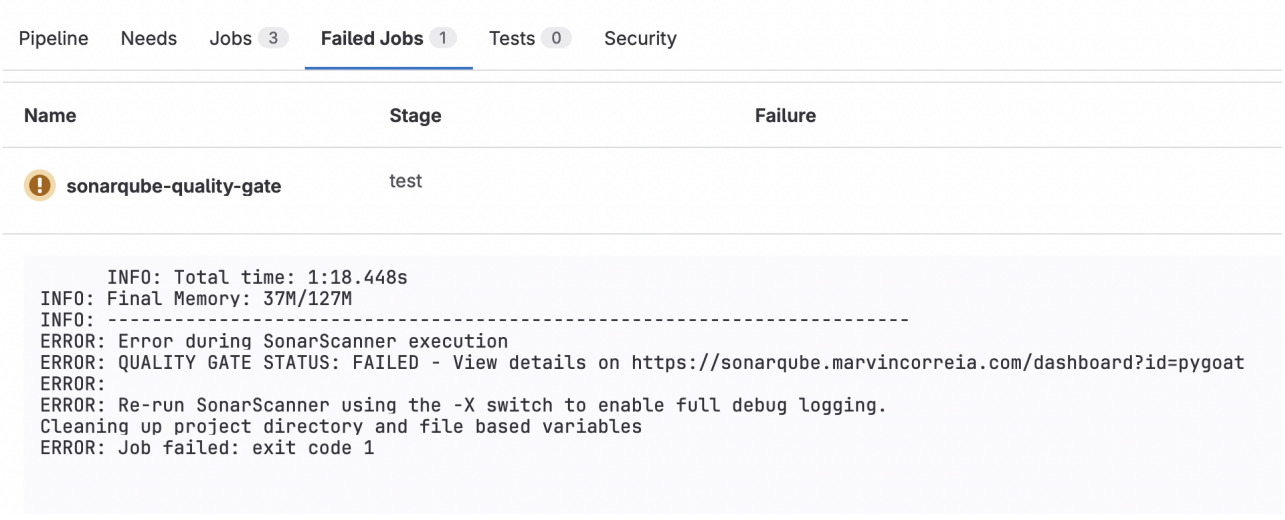

Figura 4.6: SonarQube Quality Gate error (Fuente: capt. de pantalla Gitlab)

4.4.2. SCA (Software Composition Analysis)

El análisis de composición de software es una práctica crucial en la gestión de la seguridad de proyectos de software, especialmente en aquellos que dependen en gran medida de bibliotecas y componentes de terceros. En el contexto del proyecto pygoat, el SCA se centra en examinar las dependencias utilizadas en el proyecto para detectar posibles vulnerabilidades.

Este análisis se divide en dos tipos principales de pruebas SCA:

- **Escaneo de Dependencias**: Este proceso implica escanear todas las dependencias del proyecto en busca de vulnerabilidades conocidas. Al detectar estas vulnerabilidades, los desarrolladores pueden tomar medidas correctivas antes de que el código se despliegue en producción.
- **Escaneo de Contenedor**: En esta prueba se realiza un análisis del contenedor Docker utilizado para el proyecto. Este análisis se enfoca tanto en el sistema operativo subyacente del contenedor como en las librerías y componentes instalados en él, buscando vulnerabilidades que puedan comprometer la seguridad del contenedor y, por ende, de la aplicación.

Aplicación de templates SCA

GitLab proporciona plantillas predefinidas para facilitar la implementación de estas pruebas SCA en los pipelines CI/CD. A continuación, se describe cómo incluir estas plantillas en el archivo `.gitlab-ci.yml` del proyecto para añadir los escáneres de dependencias y contenedores.

```yaml
include:
  # ...
  - template: Security/Dependency-Scanning.gitlab-ci.yml
  - template: Security/Container-Scanning.gitlab-ci.yml
```

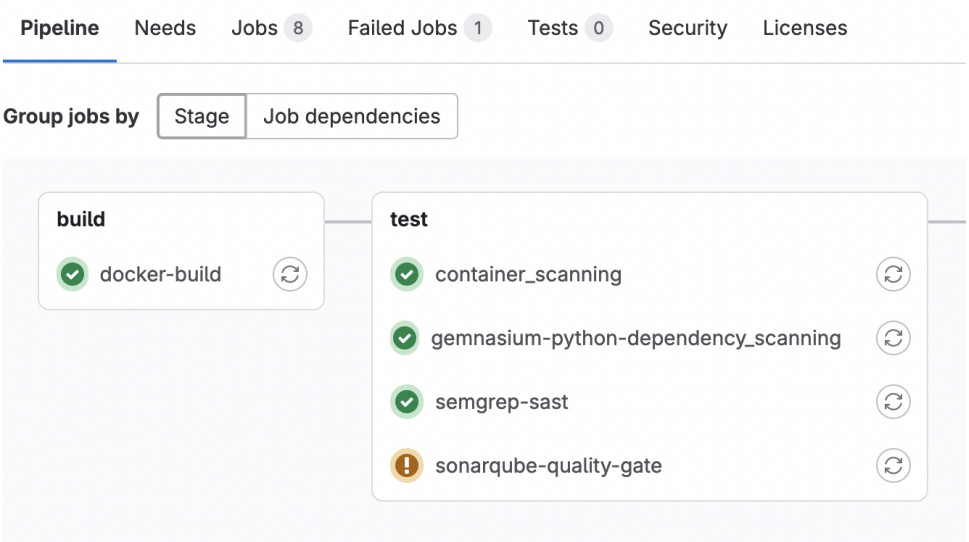

Figura 4.7: SCA Jobs (Fuente: capt. de pantalla Gitlab)

- **Plantilla de Escaneo de Dependencias**:

```
  - template: Security/Dependency-Scanning.gitlab-ci.yml
```

Esta plantilla configura automáticamente un job en el pipeline CI/CD que escanea todas las dependencias del proyecto en busca de vulnerabilidades conocidas. Utiliza bases de datos de vulnerabilidades para comparar las versiones de las dependencias del proyecto con las versiones que tienen vulnerabilidades reportadas.

- **Plantilla de Análisis del Contenedor**:

```
  - template: Security/Container-Scanning.gitlab-ci.yml
```

Esta plantilla añade un job que analiza la imagen Docker del proyecto. El escaneo revisa tanto el sistema operativo del contenedor como las librerías y componentes instalados en él, identificando vulnerabilidades que puedan afectar la seguridad de la aplicación.

4.4.3. Detección de secretos

La detección de secretos es un aspecto crucial de la seguridad en DevSecOps, ya que ayuda a identificar y prevenir la exposición de credenciales y otros datos sensibles en el código fuente. GitLab facilita este proceso mediante una plantillas específicas que pueden integrarse fácilmente en el pipeline CI/CD.

GitLeaks

GitLeaks es una herramienta ampliamente utilizada para la detección de secretos y está integrada en GitLab mediante la plantilla `Secret-Detection.gitlab-ci.yml`. Esta herramienta escanea el código en busca de patrones que puedan indicar la presencia de secretos, tales como claves de API, contraseñas y tokens.

Para configurar debemos modificar el archivo `.gitlab-ci.yml` para incluir la plantilla de detección de secretos:

```
include:
  - template: Security/Secret-Detection.gitlab-ci.yml
```

4.5. Deploy: Etapa de Despliegue

La etapa de despliegue es una fase crucial en el ciclo de vida de desarrollo y entrega de software. En esta etapa, el objetivo es tomar la imagen Docker construida en la etapa de construcción y desplegarla en un entorno de producción o de pruebas utilizando `docker-compose`. Este proceso es una parte integral de la fase de CD (entrega continua) que se enfoca en la implementación automática y consistente del software.

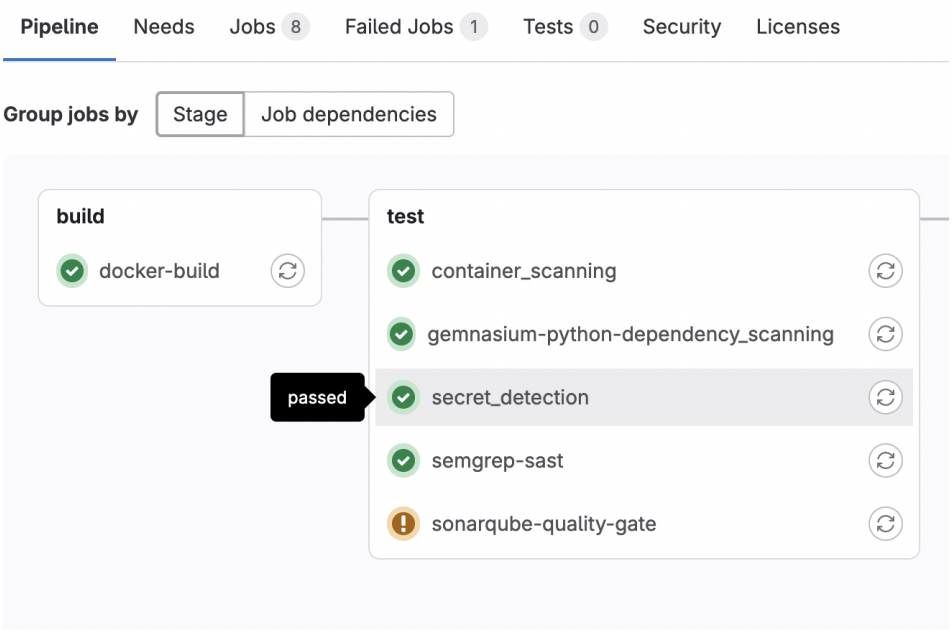

Figura 4.8: Job de Detección de secretos (Fuente: capt. de pantalla Gitlab)

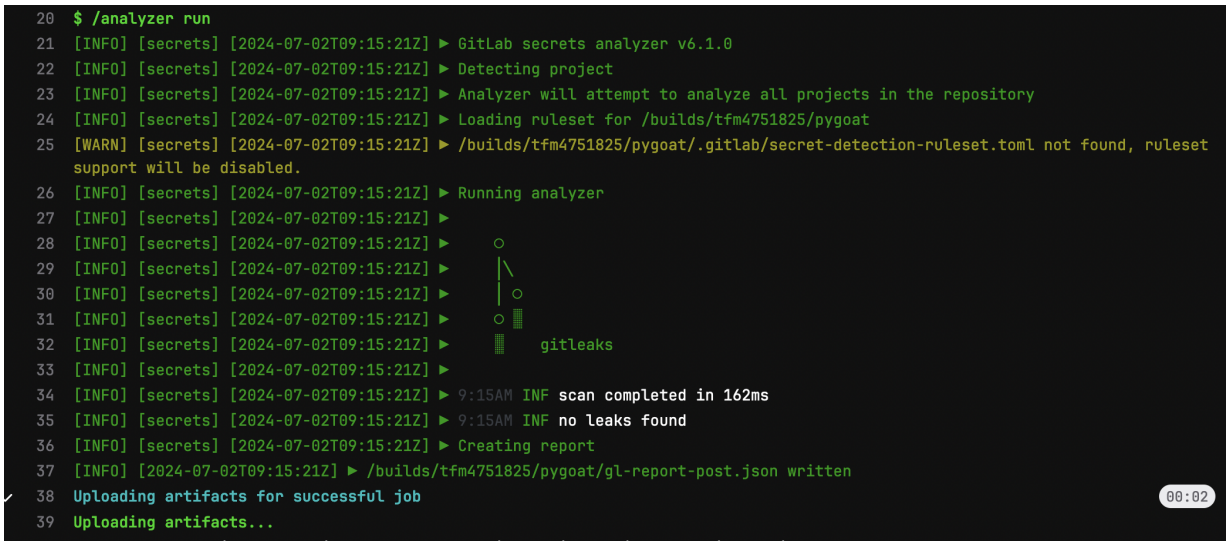

Figura 4.9: Logs del Job de Detección de secretos (Fuente: capt. de pantalla Gitlab)

4.5.1. Proceso de Despliegue

El proceso de despliegue con `docker-compose` implica varios pasos clave:

- **Preparación del Entorno**: Antes de realizar el despliegue, es necesario preparar el entorno en el que se ejecutarán los contenedores Docker. Esto incluye la configuración de los archivos `docker-compose.deploy.yml`, que describen cómo se deben configurar y ejecutar los servicios del proyecto.
- **Uso de la Imagen Construida**: La imagen Docker que se construyó en la etapa de construcción se utilizará en el archivo `docker-compose.deploy.yml`. Esta imagen contiene el proyecto empaquetado y listo para ejecutarse.
- **Despliegue con `docker-compose`**: Con el archivo `docker-compose.deploy.yml` configurado, se utiliza el comando `docker-compose up` para iniciar los servicios

definidos. Este comando crea y ejecuta los contenedores Docker según las especificaciones del archivo `docker-compose.deploy.yml`.

- **Automatización del Despliegue**: En el pipeline CI/CD, se puede automatizar este proceso mediante un *job* de GitLab CI/CD que ejecuta el despliegue como parte de la fase de CD.
- **Ejecución en GitLab Runner**: El trabajo de despliegue se ejecutará en un GitLab Runner que se instala en un servidor específico donde se servirá la aplicación que se desplegará, como muestra la figura 4.10. El GitLab Runner es un proceso que ejecuta los trabajos definidos en el pipeline CI/CD y puede ser configurado para ejecutarse en diferentes entornos, incluidos servidores dedicados, máquinas virtuales o incluso en contenedores Docker. [15]

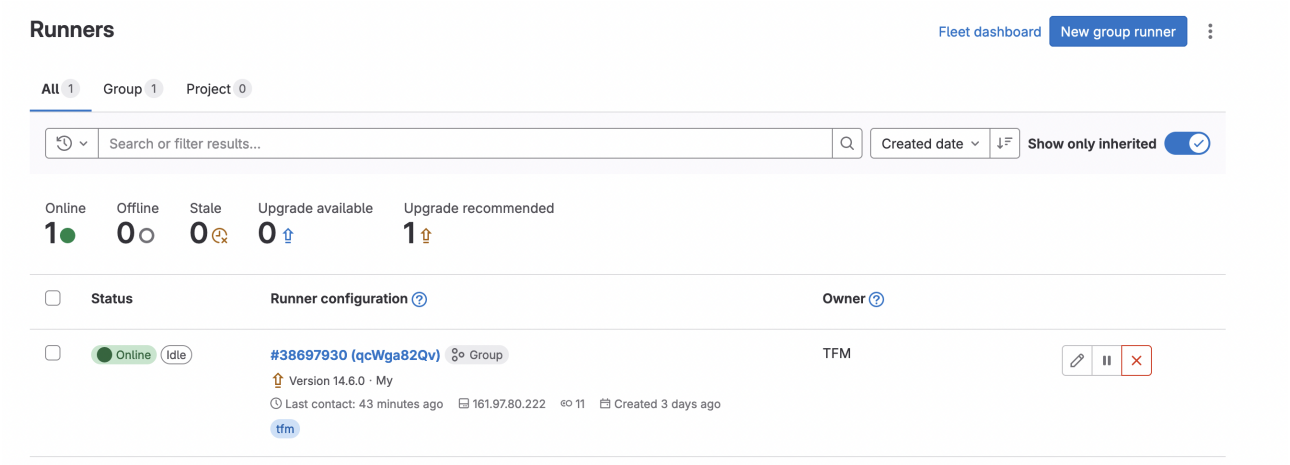

Figura 4.10: Configuración de Gitlab Runner (Fuente: capt. de pantalla Gitlab)

4.5.2. Configuración del Archivo docker-compose

El archivo `docker-compose.deploy.yml` define los servicios que forman parte de la aplicación. Aquí hay un ejemplo básico de cómo podría configurarse:

```yaml
version: "3"
services:
  web:
    image: $IMAGE:$TAG
    ports:
      - "8000:8000"
    restart: unless-stopped
    volumes:
      - /docker/pygoat/volume/:/app/volume/
```

4.5.3. Job de Despliegue en GitLab CI/CD

Para automatizar el despliegue utilizando `docker-compose` en GitLab CI/CD, se puede configurar un *job* en el archivo `.gitlab-ci.yml`:

```yaml
deploy:
  stage: deploy
  image: docker/compose
  variables:
    IMAGE: $CI_REGISTRY_IMAGE
    TAG: $CI_COMMIT_REF_SLUG
    COMPOSE_FILE_NAME: docker-compose.deploy.yml
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD
      ↪  $CI_REGISTRY
    - docker-compose -f $COMPOSE_FILE_NAME pull
    - docker-compose -f $COMPOSE_FILE_NAME -p $CI_PROJECT_NAME up -d
      ↪  --no-build
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
      when: manual
  tags:
    - tfm
```

Detalles del Job de Despliegue

- **Stage**: El *job* se ejecuta en la etapa de *deploy*.
- **Image**: Se utiliza la imagen `docker/compose` para tener acceso a los comandos de Docker Compose.
- **Variables**:
  - `IMAGE`: Define el nombre de la imagen Docker en el registro del proyecto.
  - `TAG`: Define la etiqueta de la imagen Docker basada en la rama de Git.
  - `COMPOSE_FILE_NAME`: Especifica el nombre del archivo de configuración de Docker Compose para el despliegue.
- **Script**:
  - `docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY`: Autentica en el registro de Docker.
  - `docker-compose -f $COMPOSE_FILE_NAME pull`: Descarga las imágenes Docker necesarias según la configuración del archivo `docker-compose.deploy.yml`.
  - `docker-compose -f $COMPOSE_FILE_NAME -p $CI_PROJECT_NAME up -d -no-build`: Levanta los servicios definidos en el archivo `docker-compose.deploy.yml` sin reconstruir las imágenes.
- **Tags**: Utiliza la etiqueta `tfm` para especificar los *runners* que deben ejecutar este *job*.

En este caso, hemos configurado el despliegue como un trabajo manual, y cuando ejecutamos *Run*, el despliegue se realiza, como se muestra en la figura 4.11.

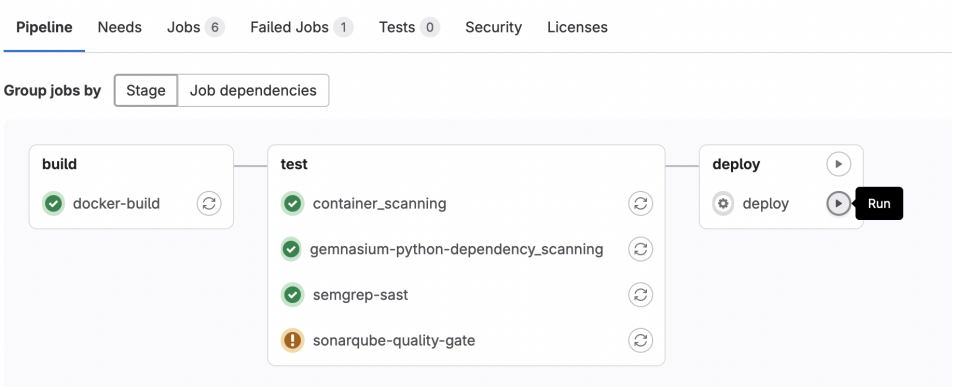

Figura 4.11: Deploy Job (Fuente: capt. de pantalla Gitlab)

Después del despliegue podemos acceder al proyecto pygoat en la web como se muestra en la imagen 4.12. Por razones de seguridad no es recomendable poner una aplicación vulnerable en exposición pública.

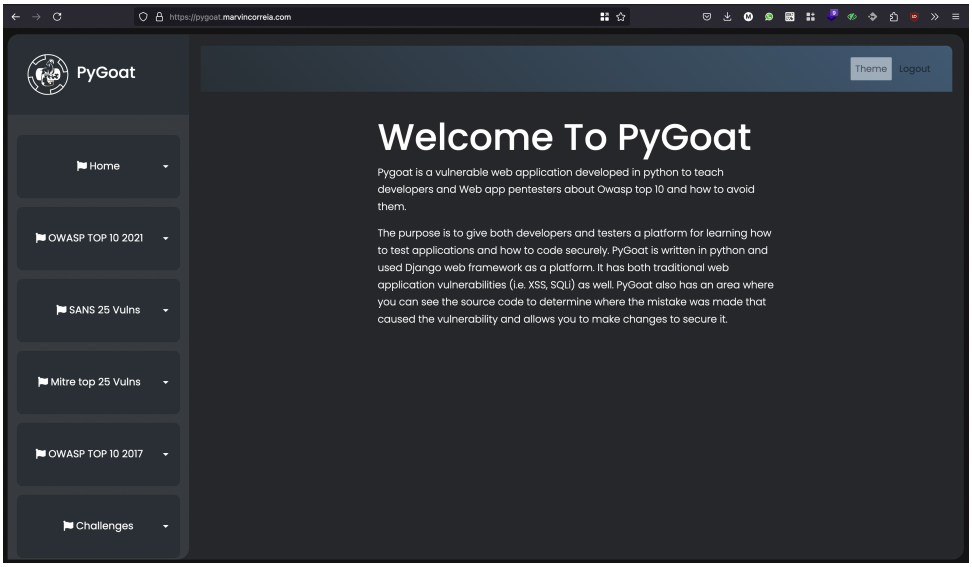

Figura 4.12: Pygoat Website en producción (Fuente: Propia)

4.6. DAST: Pruebas dinámicas de seguridad

La etapa DAST (Dynamic Application Security Testing) se centra en realizar pruebas de seguridad dinámicas en la aplicación desplegada. A diferencia de las pruebas estáticas (SAST), que analizan el código fuente, las pruebas dinámicas analizan la aplicación en ejecución para detectar posibles vulnerabilidades explotables en tiempo de ejecución. Para implementar esta etapa en el pipeline CI/CD de GitLab, utilizaremos la plantilla `gitlab-ci dast` y configuraremos un perfil DAST para especificar la URL del sitio donde se realizarán las pruebas.

4.6.1. OWASP ZAP

El escáner DAST de GitLab se basa en OWASP ZAP (Zed Attack Proxy), una herramienta popular y robusta para la evaluación de seguridad de aplicaciones web. OWASP ZAP

permite identificar vulnerabilidades comunes como inyección SQL, Cross-Site Scripting (XSS), y otras amenazas de seguridad.

4.6.2. Configuración de DAST en GitLab CI/CD

Para configurar las pruebas DAST en nuestro pipeline CI/CD, seguiremos estos pasos:

- **Incluir la plantilla DAST**: Utilizaremos la plantilla proporcionada por GitLab para añadir las pruebas DAST a nuestro pipeline.
- **Configurar el perfil DAST**: Especificaremos la URL de la aplicación desplegada donde se realizarán las pruebas de seguridad.

Incluir la Plantilla DAST

En el archivo `.gitlab-ci.yml`, incluimos la plantilla `DAST.gitlab-ci.yml` proporcionada por GitLab:

```yaml
include:
  - template: DAST.gitlab-ci.yml
```

Configuración del Perfil DAST

Para especificar la URL del sitio donde se realizará el ataque, configuramos un perfil DAST. Aquí hay un ejemplo de cómo configurar un job DAST en GitLab CI/CD:

```yaml
dast:
  stage: dast
  variables:
    DAST_AUTH_SUCCESS_IF_NO_LOGIN_FORM: "true"
  dast_configuration:
    site_profile: "pygoat"
    scanner_profile: "pygoatweb"
  when: manual
```

Al ejecutar el *job* se crea el *stage* dast como se muestra en la figura 4.13:

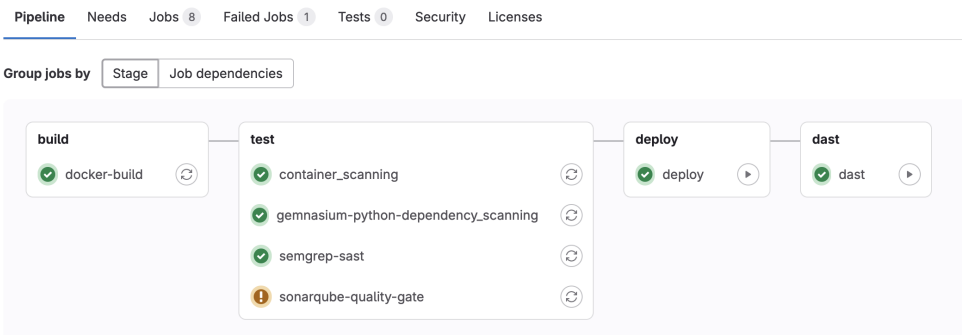

Figura 4.13: DAST Job (Fuente: capt. de pantalla Gitlab)

Detalles del Job DAST

- **Stage**: El *job* se ejecuta en la etapa `dast`.
- **DAST Configuration**:
  - `site_profile`: Define el perfil del sitio que contiene la URL y otras configuraciones, como muestra la figura 4.14.
  - `scanner_profile`: Define el perfil del escáner con configuraciones específicas para OWASP ZAP.

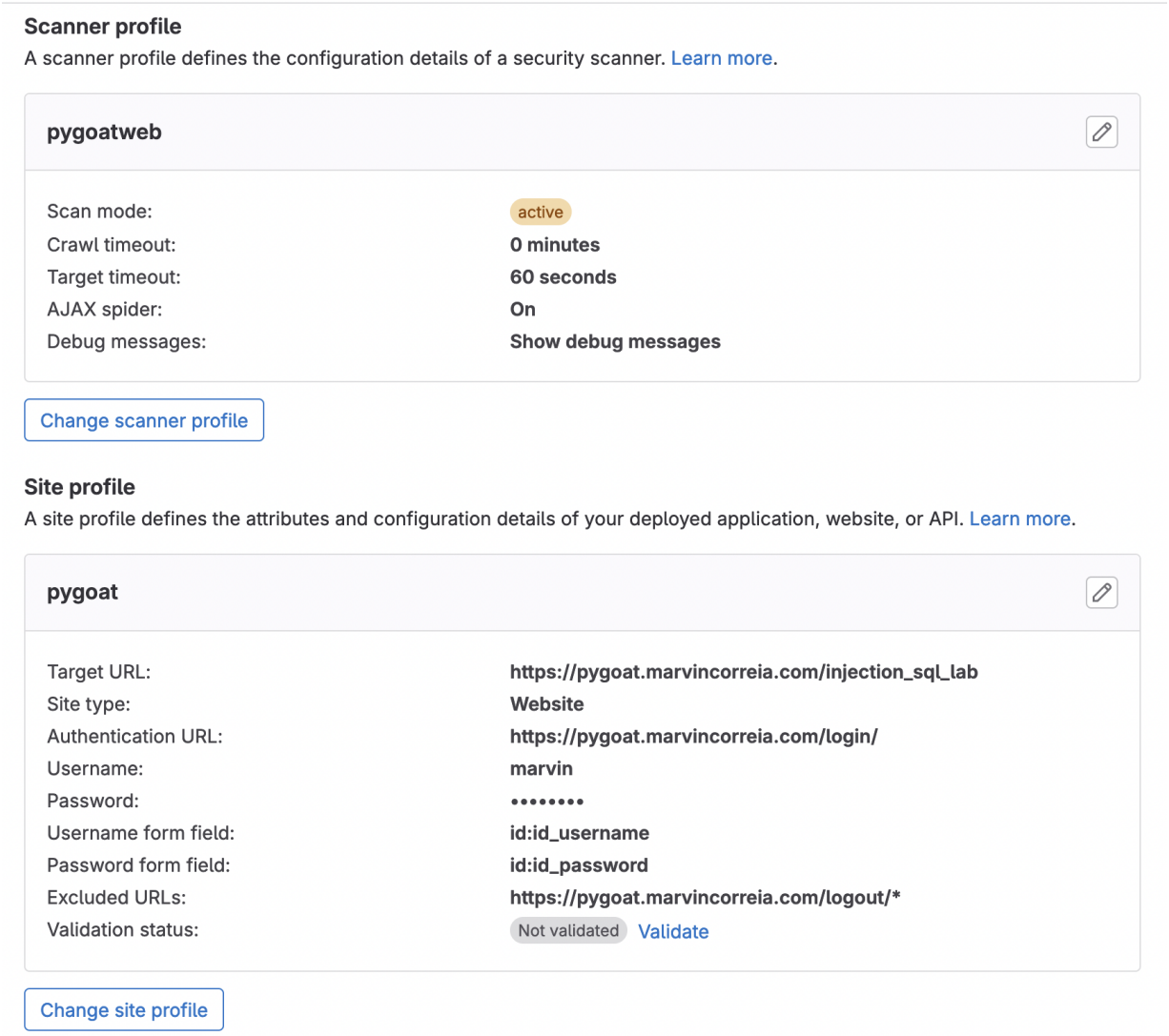

Figura 4.14: DAST Scanner / Site profile (Fuente: capt. de pantalla Gitlab)

Validación del Sitio para Pruebas DAST

Dado que las pruebas DAST implican ataques activos contra la aplicación, es fundamental asegurarse de que estos ataques se dirigen a un sitio validado y autorizado para dichas pruebas. Para evitar que las pruebas DAST afecten inadvertidamente a sistemas no autorizados, es esencial realizar una validación del sitio antes de ejecutar las pruebas. En nuestro caso, utilizaremos la validación de header en el servidor web para confirmar que el sitio está autorizado para las pruebas DAST.

Configuración de Validación de Header

Para garantizar que las pruebas DAST se realizan en un sitio autorizado, configuraremos Nginx [23] como un servidor reverse proxy que añada un header de validación específica a todas las respuestas HTTP. Este header será verificada antes de que se ejecuten las pruebas DAST. La figura 4.15 muestra como podemos hacer la configuración en Gitlab.

```
server {
    server_name pygoat.marvincorreia.com;
    listen 443 ssl; # managed by Certbot
    ...
    location / {
        proxy_set_header Host $host;
        add_header Gitlab-On-Demand-DAST
        ↪  "5ea12b8d-a588-4c6a-94cc-2e4db64bab40";
        proxy_pass http://localhost:8000;
    }
}
```

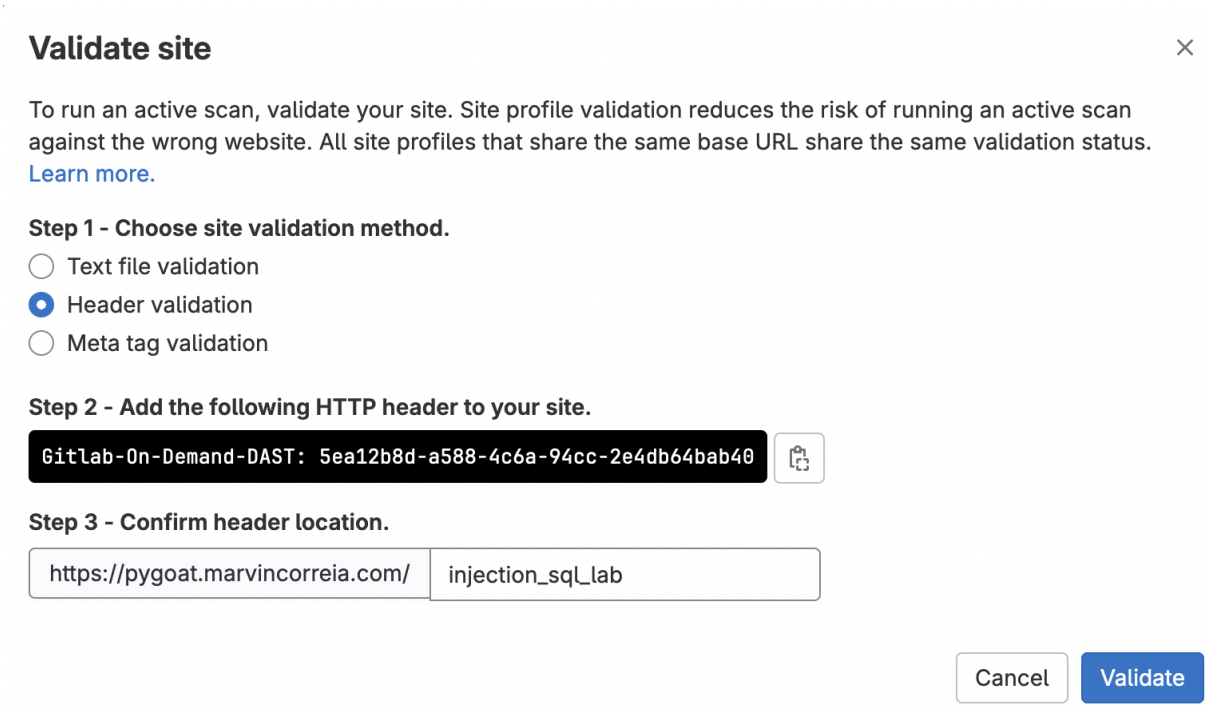

Figura 4.15: Validación del Sitio para Pruebas DAST (Fuente: capt. de pantalla Gitlab)

4.7. Gestión de Vulnerabilidades

En el desarrollo de software, la gestión de vulnerabilidades es un aspecto crucial para garantizar la seguridad y la integridad del sistema. GitLab ofrece herramientas integradas que permiten a los equipos de desarrollo y seguridad gestionar eficazmente las vulnerabilidades que se detectan durante las etapas de CI/CD. En este apartado,

abordaremos cómo gestionar las vulnerabilidades encontradas y cómo GitLab proporciona un entorno centralizado para esta tarea.

4.7.1. Informe de Vulnerabilidades

El Informe de Vulnerabilidades de GitLab en 4.16, es una herramienta centralizada que permite a los equipos visualizar y gestionar las vulnerabilidades detectadas en sus proyectos. Este panel ofrece una visión general de todas las vulnerabilidades, clasificándolas por severidad y estado, lo que facilita la priorización y la toma de decisiones sobre las acciones correctivas necesarias.

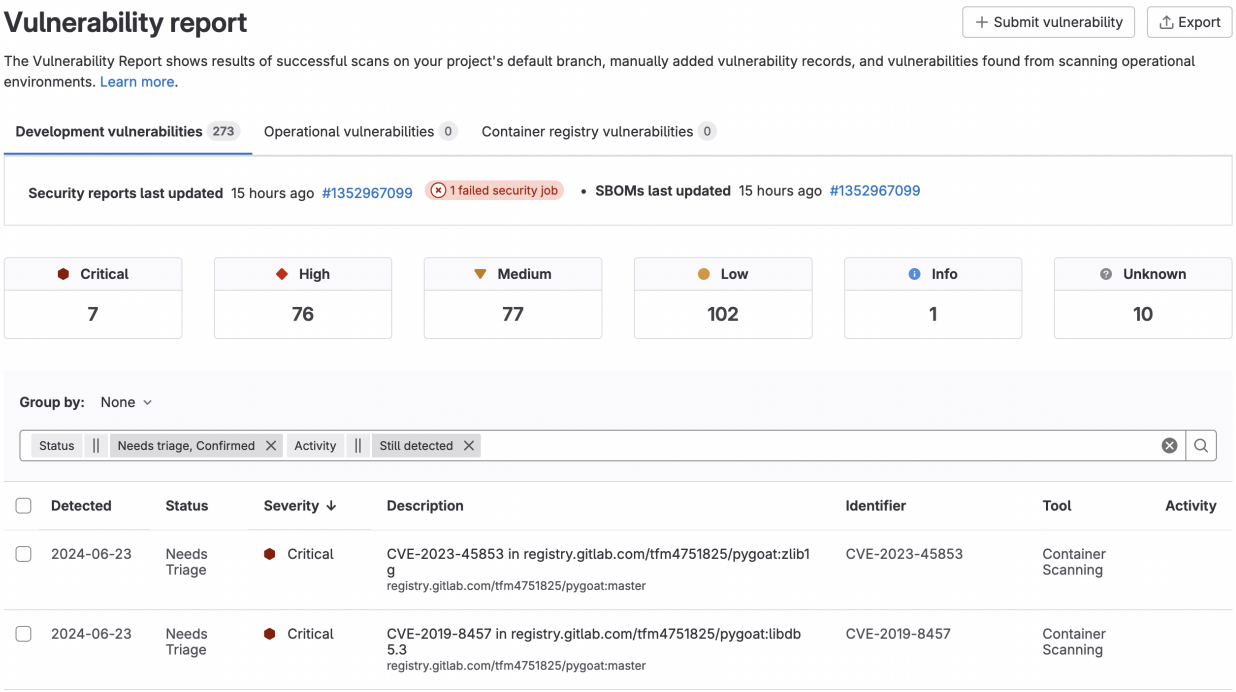

Figura 4.16: Informe de Vulnerabilidades (Fuente: capt. de pantalla Gitlab)

- **Visión General**: Proporciona un resumen de todas las vulnerabilidades detectadas en la rama por defecto del proyecto.
- **Clasificación por Severidad**: Las vulnerabilidades se clasifican como críticas, altas, medias o bajas, lo que ayuda a priorizar las acciones correctivas.
- **Estado de las Vulnerabilidades**: Permite a los usuarios ver el estado actual de cada vulnerabilidad (por ejemplo, abierta, resuelta, ignorada).
- **Detalles Específicos**: Para cada vulnerabilidad, el informe incluye información detallada como la descripción, la ubicación en el código, la recomendación para la corrección y la severidad.
- **Filtrado y Ordenación**: Los usuarios pueden filtrar y ordenar las vulnerabilidades por diversos criterios, como severidad, tipo y estado.
- **Historial de Cambios**: Muestra el historial de quién realizó los cambios.

4.7.2. Dashboard de Seguridad

El Panel de Seguridad en GitLab, como se muestra en la figura 4.17, permite a los equipos visualizar la evolución de las vulnerabilidades detectadas en sus proyectos. Este panel ofrece un gráfico que muestra cómo las vulnerabilidades han cambiado en cantidad y severidad.

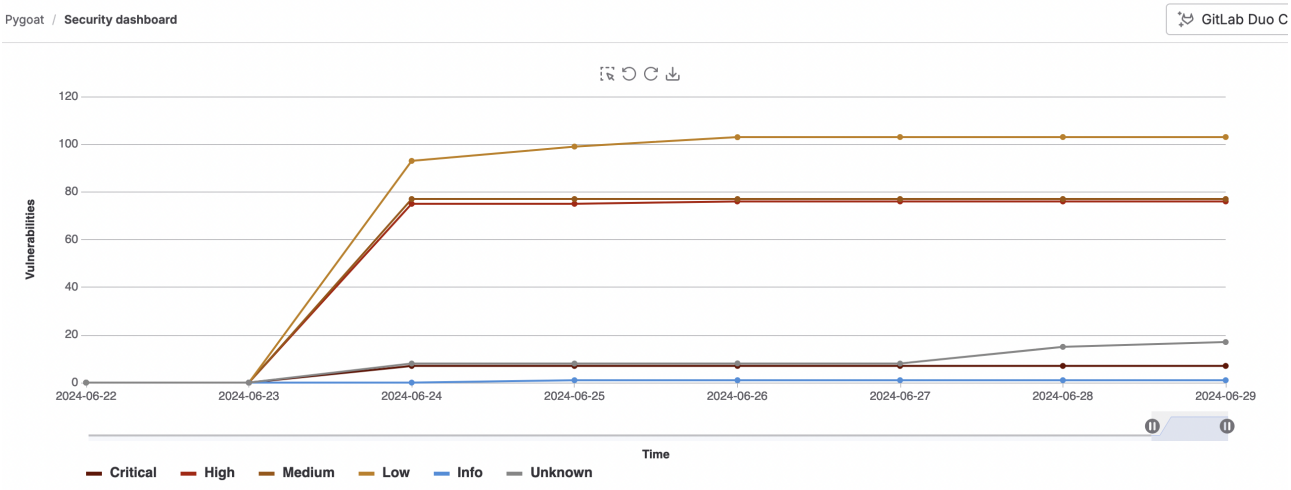

Figura 4.17: Dashboard de Seguridad (Fuente: capt. de pantalla Gitlab)

- Proporciona un gráfico que muestra la evolución a lo largo del tiempo de las vulnerabilidades detectadas en la rama por defecto del proyecto.
- Las vulnerabilidades se clasifican como críticas, altas, medias o bajas, lo que ayuda a priorizar las acciones correctivas.

Es importante destacar que los datos que se muestran en el Dashboard de Seguridad y en el Informe de Vulnerabilidades se limitan a la rama por defecto del proyecto. Esto asegura que las vulnerabilidades más relevantes y críticas para el estado actual del proyecto sean las que reciban atención prioritaria. Esta focalización en la rama por defecto garantiza que los equipos trabajen siempre en la versión más segura del software, priorizando la resolución de problemas en la línea de código principal antes de considerar otras ramas o versiones.

Capítulo 5

Resultados y Discusión

En este capítulo, presentaremos las vulnerabilidades detectadas a lo largo de los diferentes trabajos de seguridad que hemos implementado en el pipeline CI/CD. Estos resultados son cruciales para entender el estado de seguridad de nuestra aplicación en cuestión y tomar las acciones necesarias para mitigar cualquier riesgo identificado.

El pipeline final presentado en la figura 5.1 muestra el conjunto de trabajos ejecutados en cada *stage*, y en la figura 5.2 se muestran las vulnerabilidades encontradas durante el proceso.

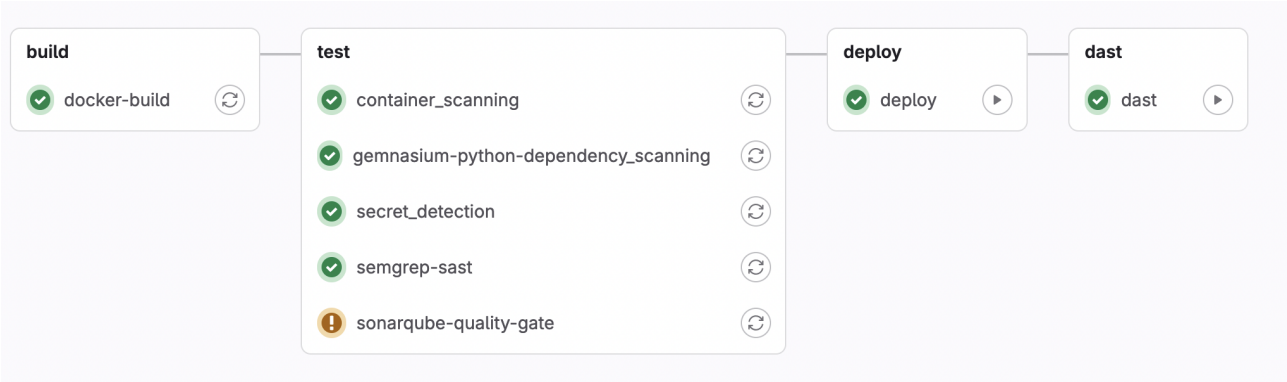

Figura 5.1: Pipeline Final (Fuente: capt. de pantalla Gitlab)

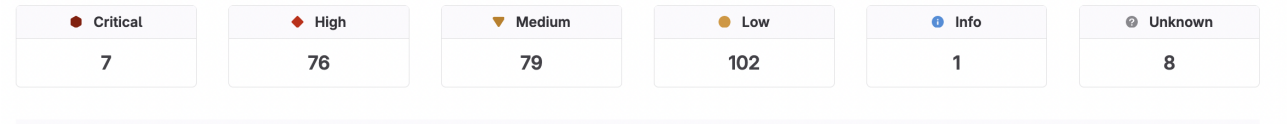

Figura 5.2: Distribución de vulnerabilidades por severidad (Fuente: capt. de pantalla Gitlab)

En la imagen 5.3, se muestran los resultados de diversas pruebas de seguridad realizadas a través del pipeline de CI/CD en GitLab. A continuación, se detallan los hallazgos de cada tipo de prueba:

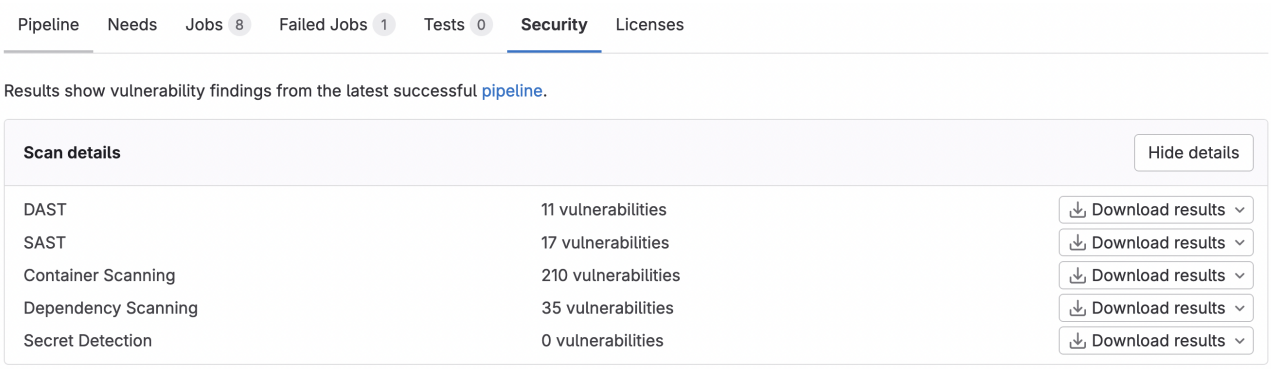

Figura 5.3: Total de Vulnerabilidades Encontradas (Fuente: capt. de pantalla Gitlab)

5.1. Vulnerabilidades Detectadas en SAST

El análisis de seguridad estática (SAST) se ejecutó durante la etapa de pruebas y generó un informe detallado de las vulnerabilidades encontradas en el código fuente, como muestra la imagen 5.4. A continuación se presentan algunas de las vulnerabilidades detectadas:

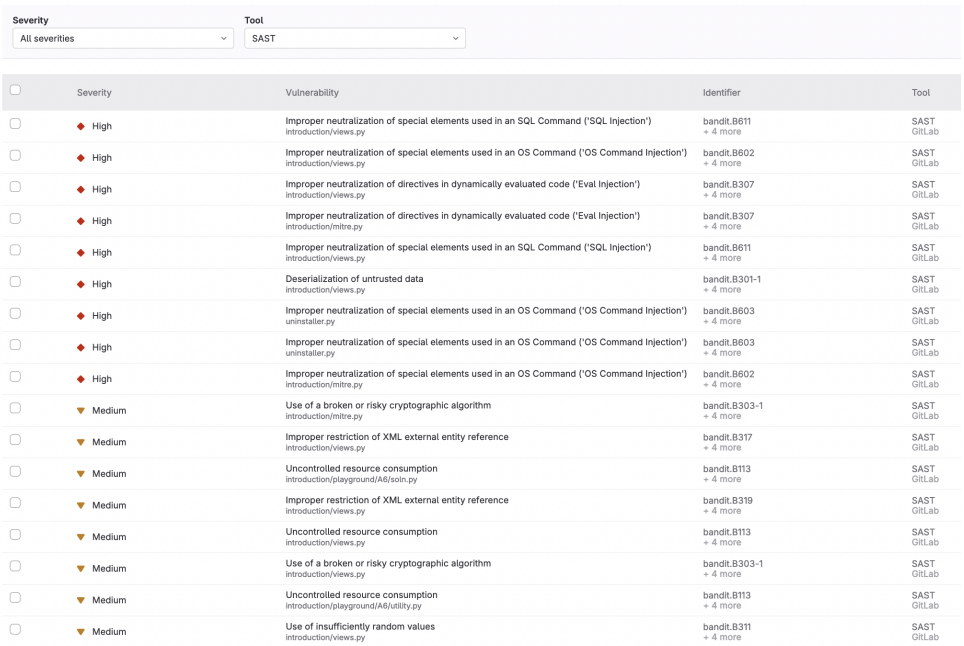

Figura 5.4: Vulnerabilidades en SAST (Fuente: capt. de pantalla, Gitlab)

- **Inyección SQL**: Se detectaron varias instancias donde las consultas SQL eran construidas utilizando concatenación de cadenas, lo cual es susceptible a inyecciones SQL.
- **Uso de un algoritmo criptográfico débil**: Se ha descubierto que la aplicación utiliza algoritmos de hash débil o arriesgado (MD2, MD4, MD5 y SHA1).
- **Neutralización incorrecta de elementos especiales utilizados en un comando OS ('OS Command Injection')**: se encontró la función de subproceso Popen con shell=True

- **Neutralización inapropiada de directivas en código evaluado dinámicamente (‘Eval Injection’)**: se encontró la aplicación llamando a la función eval con datos no literales.

Resultados del Análisis en SonarQube

La imagen 5.5 muestra un análisis detallado del código, evaluando múltiples aspectos de calidad como seguridad, confiabilidad y mantenibilidad en la herramienta Sonarqube. En cuanto a seguridad, se han identificado 2 problemas, clasificados todos como de alta gravedad (2 H).

En términos de confiabilidad, se han encontrado 30 problemas. Estos problemas están distribuidos en diferentes niveles de gravedad: 7 de alta, 20 de mediana y 3 de baja. La mantenibilidad también presenta desafíos, con un total de 313 problemas. La mayor parte de estos problemas son de alta gravedad (124 H).

Otros aspectos destacados incluyen una duplicación del código del 8.6 % en 13,000 líneas de código nuevas y la cobertura de pruebas, que actualmente es del 0 % sobre 1,700 líneas de código. Además, se han identificado 163 "security hotspots", que requieren atención para garantizar la seguridad del código.

Estos resultados ofrecen una visión clara de las áreas que necesitan mejoras para asegurar la calidad y seguridad del software.

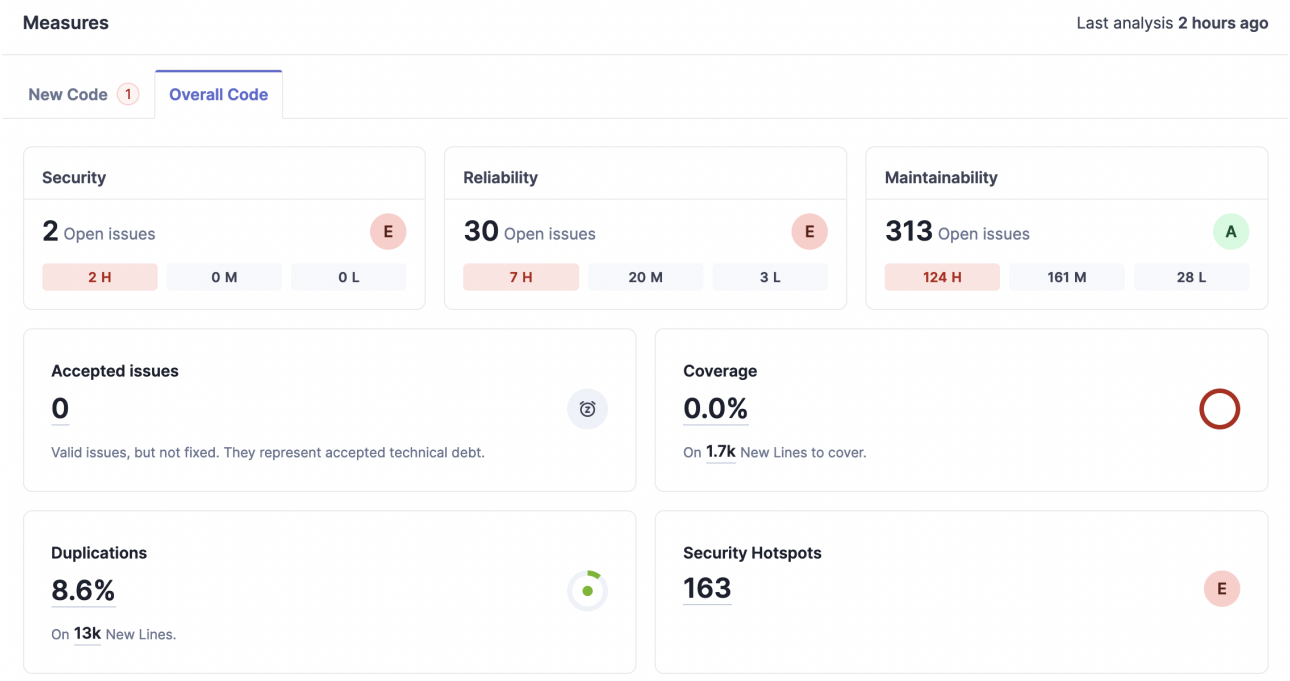

Figura 5.5: Resultados de Sonaqube (Fuente: capt. de pantalla, Sonarqube Self-Hosted)

5.2. Vulnerabilidades Detectadas en SCA

El análisis de composición de software (SCA) se centró en las dependencias utilizadas en el proyecto, escaneando en busca de vulnerabilidades conocidas en las bibliotecas, componentes de terceros y contenedores, como muestra las figuras 5.6 y 5.7 . Los resultados son los siguientes:

Vulnerabilidades de dependencias

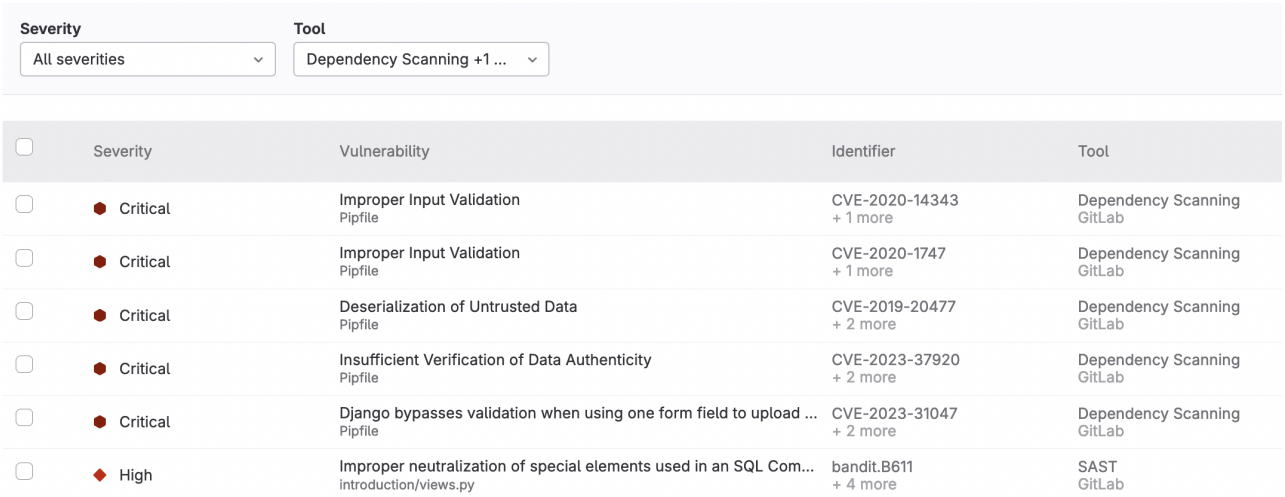

Figura 5.6: Vulnerabilidades de dependencias (Fuente: capt. de pantalla, Gitlab)

- **Validación de Entrada Inadecuada - PyYaml**: Se ha descubierto una vulnerabilidad en la librería PyYAML, donde es susceptible de ejecución de código arbitrario cuando procesa archivos YAML no confiables a través del método full_load o con el cargador FullLoader.
- **Django omite la validación cuando se utiliza un campo de formulario para subir varios archivos**: En Django 3.2 antes de 3.2.19, 4.x antes de 4.1.9 y 4.2 antes de 4.2.1, era posible saltarse la validación al utilizar un campo de formulario para subir varios archivos.

Vulnerabilidades de Contenedor

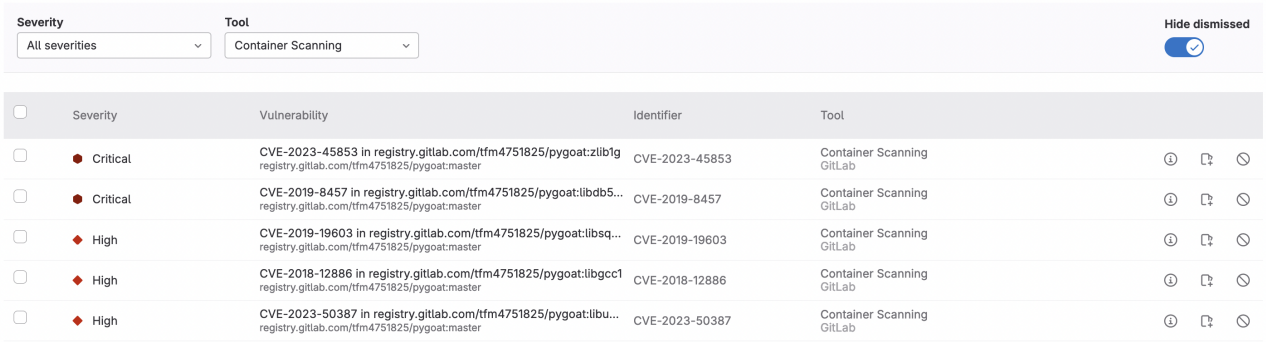

Figura 5.7: Vulnerabilidades de Contenedor (Fuente: capt. de pantalla, Gitlab)

- **CVE-2023-45853 - zlib1g**: MiniZip en zlib a través de 1.3 tiene un *buffer overflow* basado en heap en zipOpenNewFileInZip4_64 a través de un nombre de archivo largo.
- **CVE-2019-8457 - libdb5.3**: SQLite3 desde la versión 3.6.0 hasta la 3.27.2 inclusive es vulnerable a la lectura fuera de límites de heap en la función rtreenode() al manejar tablas rtree no válidas.

5.3. Vulnerabilidades Detectadas en DAST

La siguiente imagen 5.8, muestra la lista de vulnerabilidades encontradas en las pruebas DAST:

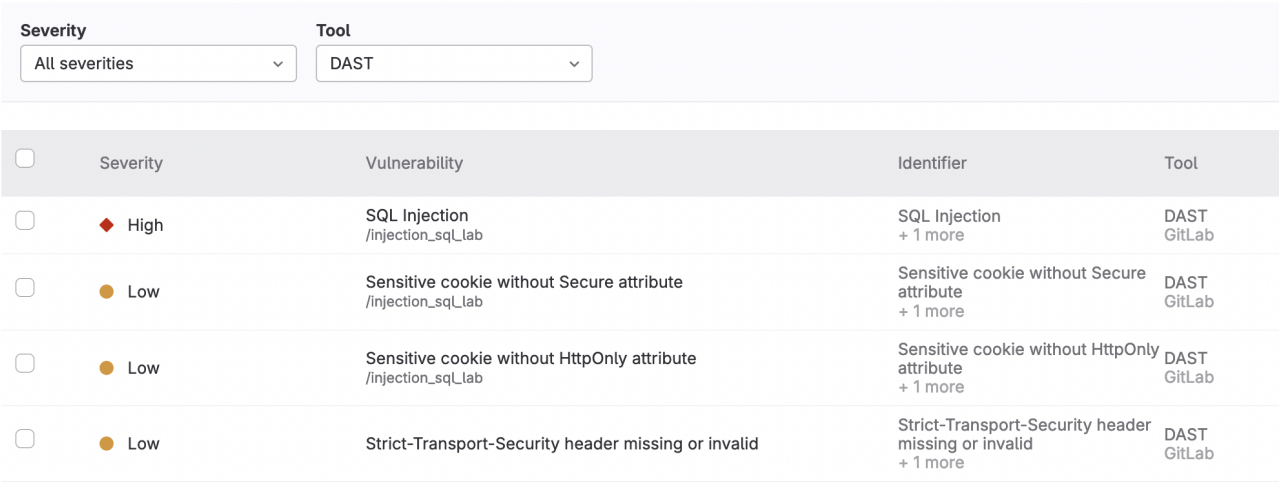

Figura 5.8: Vulnerabilidades Detectadas en DAST (Fuente: capt. de pantalla, Gitlab)

- **SQL Injection**: Es posible ejecutar comandos SQL arbitrarios en la base de datos backend del servidor de aplicaciones objetivo. La inyección SQL es una vulnerabilidad crítica que puede comprometer los datos o el sistema en `/injection_sql_lab`.
- **Falta el header Strict-Transport-Security o no es válida**: Se ha detectado que el encabezado Strict-Transport-Security falta o no es válido. El encabezado Strict-Transport-Security permite a los operadores de sitios web forzar que las comunicaciones se produzcan a través de una conexión TLS.

Capítulo 6

Conclusiones y líneas futuras

En este trabajo, se ha abordado la integración de prácticas de DevSecOps en pipelines de CI/CD, con un enfoque particular en la plataforma GitLab. A través de un análisis exhaustivo de conceptos fundamentales, herramientas y tecnologías disponibles, se ha desarrollado un pipeline robusto que incluye pruebas de seguridad estáticas, análisis de composición de software y detección de secretos, así como pruebas dinámicas de seguridad. Los resultados obtenidos demuestran la eficacia de estas prácticas en la identificación y mitigación de vulnerabilidades, mejorando significativamente la seguridad del ciclo de vida del software.

6.1. Conclusión

6.1.1. Integración Eficaz de DevSecOps

La implementación de un pipeline de CI/CD con prácticas de DevSecOps en GitLab ha permitido una integración fluida de seguridad que puede ser implementada desde las etapas iniciales del desarrollo. Esto facilita la detección temprana de vulnerabilidades y la aplicación de medidas correctivas sin comprometer la agilidad del proceso.

6.1.2. Resultados Positivos en la Gestión de Vulnerabilidades

Los resultados obtenidos, detallados en el capítulo de Resultados y Discusión, evidencian la capacidad del pipeline para identificar un amplio espectro de vulnerabilidades en diferentes fases del ciclo de vida del software. Herramientas de pruebas SAST, SCA y DAST han sido fundamentales en este proceso, proporcionando una cobertura completa de seguridad.

6.1.3. Importancia de la Automatización

La automatización de pruebas de seguridad y la integración continua han demostrado ser componentes críticos para mantener un alto nivel de seguridad sin ralentizar el desarrollo. La automatización no solo mejora la eficiencia, sino que también reduce el riesgo de errores humanos y omisiones en la revisión manual.

6.2. Líneas Futuras

6.2.1. Ampliación de Herramientas y Tecnologías

Es fundamental seguir explorando y adoptando nuevas herramientas y tecnologías que emerjan en el campo de DevSecOps. Esto incluye la integración de nuevas soluciones de análisis estático y dinámico, así como herramientas de monitoreo y respuesta a incidentes en tiempo real.

6.2.2. Mejora Continua del Pipeline

El pipeline debe ser objeto de una mejora continua, ajustándose a las necesidades cambiantes del proyecto y a las nuevas amenazas de seguridad que puedan surgir. Esto incluye la actualización regular de las herramientas de seguridad y la incorporación de nuevas prácticas recomendadas por la comunidad de seguridad.

6.2.3. Implementación de Políticas de Seguridad Rigurosas

Desarrollar e implementar políticas de seguridad rigurosas que se integren de manera natural en el flujo de trabajo del pipeline. Esto incluye la gestión segura de secretos y credenciales, la implementación de controles de acceso estrictos y la conformidad con regulaciones y estándares de la industria.

6.2.4. Análisis de Impacto y Rendimiento

Realizar análisis regulares del impacto y rendimiento del pipeline de seguridad para identificar áreas de mejora y optimización. Esto incluye la evaluación del tiempo de ejecución de las pruebas de seguridad y su impacto en el ciclo de desarrollo.

6.3. Agradecimientos

Esta investigación resulta de del Proyecto Estratégico SCITALA (C064/23), fruto del convenio de colaboración suscrito entre el Instituto Nacional de Ciberseguridad (INCIBE) y la Universidad de La Laguna. Esta iniciativa se realiza en el marco de los fondos del Plan de Recuperación, Transformación y Resiliencia, financiados por la Unión Europea (Next Generation).

Capítulo 7

Summary and conclusions

In this work, the integration of DevSecOps practices in CI/CD pipelines has been addressed, with a particular focus on the GitLab platform. Through an exhaustive analysis of fundamental concepts, available tools, and technologies, a robust pipeline has been developed that includes static security testing, software composition analysis, secret detection, and dynamic security testing. The obtained results demonstrate the effectiveness of these practices in identifying and mitigating vulnerabilities, significantly enhancing the security of the software lifecycle.

7.1. Conclusion

7.1.1. Effective Integration of DevSecOps

The implementation of a CI/CD pipeline with DevSecOps practices in GitLab has enabled a smooth integration of security that can be implemented from the early stages of development. This facilitates early detection of vulnerabilities and the application of corrective measures without compromising the agility of the process.

7.1.2. Positive Results in Vulnerability Management

The results obtained, detailed in the Results and Discussion chapter, highlight the pipeline's ability to identify a broad spectrum of vulnerabilities at different phases of the software lifecycle. SAST, SCA, and DAST testing tools have been fundamental in this process, providing comprehensive security coverage.

7.1.3. Importance of Automation

The automation of security testing and continuous integration have proven to be critical components for maintaining a high level of security without slowing down development. Automation not only improves efficiency but also reduces the risk of human errors and omissions in manual review.

Bibliografía

[1] *Aikido*. https://www.aikido.dev. 2024.

[2] Anchore. *What is DevSecOps? A Guide for 2024*. 2024. url: https://anchore.com/devsecops/what-is-devsecops/.

[3] *Aqua Security*. https://www.aquasec.com. 2024.

[4] Atlassian. *5 Best CI/CD Tools Every DevOps Needs*. 2024. url: https://www.atlassian.com/devops/devops-tools/cicd-tools.

[5] AWS. *Building end-to-end AWS DevSecOps CI/CD pipeline with open source SCA, SAST and DAST tools*. Accessed: 2024-06-26. 2021. url: https://aws.amazon.com/blogs/devops/building-end-to-end-aws-devsecops-ci-cd-pipeline-with-open-source-sca-sast-and-dast-tools/.

[6] Jim Bird. *DevSecOps: A leader's guide to producing secure software without compromising flow, feedback and continuous improvement*. Sebastopol, CA: O'Reilly Media, 2019. isbn: 978-1492044295.

[7] *Checkmarx*. https://checkmarx.com. 2024.

[8] Azure DevOps. *Enable DevSecOps with Azure and GitHub*. Accessed: 2024-06-26. 2022. url: https://github.com/adeyosemanputra/pygoathttps://learn.microsoft.com/en-us/devops/devsecops/enable-devsecops-azure-github.

[9] eSecurityPlanet. *15 Best DevSecOps Tools For Seamless Security In 2024*. 2024. url: https://www.esecurityplanet.com/products/devsecops-tools/.

[10] Nicole Forsgren, Jez Humble y Gene Kim. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. Portland, OR: IT Revolution Press, 2018. isbn: 978-1942788331.

[11] Inc. Gartner. *Gartner Says Cybersecurity Risk Set to Be a Primary Buying Consideration for Chief Supply Chain Officers*. https://www.gartner.com/en/newsroom/press-releases/2023-03-30-gartner-says-cybersecurity-risk-set-to-be-a-primary-buying-consideration-for-chief-supply-chain-officers. Accessed: 2024-07-03. Mar. de 2023.

[12] GitLab. *GitLab CI Templates*. Accessed: 2024-06-26. 2024. url: https://gitlab.com/gitlab-org/gitlab-foss/-/tree/master/lib/gitlab/ci/templates.

[13] GitLab. *GitLab CI/CD Pipelines*. Accessed: 2024-07-02. 2024. url: https://docs.gitlab.com/ee/ci/pipelines/.

[14] GitLab. *Security and Compliance Solutions*. 2024. url: https://about.gitlab.com/solutions/security-compliance/#demo.

[15] Gitlab. *Gitlab Runner*. url: https://docs.gitlab.com/runner/.

[16] *GitLab CI/CD*. https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/. 2024.

[17] Basana Gouda. *Day 1: Getting Started with DevOps*. Accessed: 2024-07-02. 2023. url: https://basanagouda.medium.com/day-1-getting-started-with-devops-b78ccca0af8.

[18] hackbysecurity. *CVE, CWE, CAPEC, CVSS, vaya lío...* 2022. url: https://www.hackbysecurity.com/blog/cve-cwe-capec-cvss-vaya-lio.

[19] Expert Insights. *The Top Static Application Security Testing (SAST) Tools*. Accessed: 2024-07-02. 2024. url: https://expertinsights.com/insights/the-top-static-application-security-testing-sast-tools/.

[20] Gene Kim et al. *The DevOps Handbook: How to Create World-Class Agility, Reliability, Security in Technology Organizations*. Portland, OR: IT Revolution Press, 2016. isbn: 978-1942788003.

[21] Mindbowser. *CI/CD Pipeline Tools*. Accessed: 2024-07-02. 2024. url: https://www.mindbowser.com/ci-cd-pipeline-tools/.

[22] The Hacker News. *What is DevSecOps and Why is it Essential for Secure Software Delivery?* https://thehackernews.com/2024/06/what-is-devsecops-and-why-is-it.html. Accessed: 2024-07-03. Jun. de 2024.

[23] Nginx. *Nginx reverse proxy*. url: https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/.

[24] OWASP Foundation. *OWASP Attacks*. 2024. url: https://owasp.org/www-community/attacks/.

[25] OWASP Foundation. *OWASP Vulnerabilities*. 2024. url: https://owasp.org/www-community/vulnerabilities/.

[26] *OWASP ZAP*. https://www.zaproxy.org. 2024.

[27] Ade Yoseman Putra. *PyGoat: Intentionally vulnerable web Application Security in django*. Accessed: 2024-06-26. 2023. url: https://github.com/adeyosemanputra/pygoat.

[28] Mary Sánchez-Gordón y Ricardo Colomo-Palacios. "Security as culture: a systematic literature review of DevSecOps". En: *Proceedings of the IEEE/ACM 42nd international conference on software engineering workshops*. 2020, págs. 266-269.

[29] Semgrep. *Semgrep Code*. Accessed: 2024-06-26. 2024. url: https://semgrep.dev/products/semgrep-code.

[30] Kelly Shortridge y Aaron Rinehart. *Security Chaos Engineering: Sustaining Resilience in Software and Systems*. Sebastopol, CA: O'Reilly Media, 2020. isbn: 978-1492056465.

[31] *Snyk*. https://snyk.io. 2024.

[32] Perforce Software. *What is Shift Left Security?* 2024. url: https://www.perforce.com/blog/sca/what-is-shift-left-security.

[33] *SonarQube*. https://www.sonarqube.org. 2024.

[34] StationX. *25 Top DevSecOps Tools (Ultimate Guide for 2024)*. 2024. url: https://www.stationx.net/top-devsecops-tools/.

[35] Cloud Tips. *DevSecOps: Definition, Best Practices, and Tools*. Accessed: 2024-07-02. 2023. url: https://medium.com/@cloud_tips/devsecops-definition-best-practices-and-tools-1789587d165a.

[36] *Veracode*. https://www.veracode.com. 2024.

Apéndice A

Configuración del Pipeline CI/CD completo

```yaml
# Definición de los stages (etapas) del pipeline
stages:
  - build
  - test
  - deploy
  - dast

include:
  - template: Security/SAST.gitlab-ci.yml
  - template: Security/Dependency-Scanning.gitlab-ci.yml
  - template: Security/Container-Scanning.gitlab-ci.yml
  - template: Security/Secret-Detection.gitlab-ci.yml
  - template: Security/DAST.gitlab-ci.yml

# BUILD JOB: build docker image and store in registry
docker-build:
  stage: build
  image: docker:cli
  services:
    - docker:dind
  variables:
    DOCKER_IMAGE_NAME: $CI_REGISTRY_IMAGE:$CI_COMMIT_REF_SLUG
  before_script:
    - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD"
      ↪  $CI_REGISTRY
  script:
    - docker build --pull -t "$DOCKER_IMAGE_NAME" .
    - docker push "$DOCKER_IMAGE_NAME"
  # Run this job in a branch where a Dockerfile exists
  rules:
    - if: $CI_COMMIT_BRANCH
      exists:
        - Dockerfile
  needs: []

# SAST JOBS
```

```yaml
# override original gitlab sast job, add "needs" to allow running in paralel with build job
sast:
  needs: []

secret_detection:
  needs: []

# sonarqube: check code quality and vulnerabilities
sonarqube-quality-gate:
  stage: test
  image:
    name: sonarsource/sonar-scanner-cli:5.0
    entrypoint: [""]
  variables:
    SONAR_USER_HOME: "${CI_PROJECT_DIR}/.sonar" # Defines the location of the analysis task cache
    GIT_DEPTH: "0" # Tells git to fetch all the branches of the project, required by the analysis task
  cache:
    key: "${CI_JOB_NAME}"
    paths:
      - .sonar/cache
  script:
    - sonar-scanner
  allow_failure: true
  needs:
    - []
  only:
    - merge_requests
    - master
    - main
    - develop
    - dev


# SCA JOBS (Software composition analyses)
dependency_scanning:
  needs: []

container_scanning:
  variables:
    CS_IMAGE: $CI_REGISTRY_IMAGE:$CI_COMMIT_REF_SLUG
  needs:
    - docker-build

# Deploy Stage Job
deploy:
```

```
  stage: deploy
  image: docker/compose
  variables:
    IMAGE: $CI_REGISTRY_IMAGE
    TAG: $CI_COMMIT_REF_SLUG
    COMPOSE_FILE_NAME: docker-compose.deploy.yml
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD
    ↪  $CI_REGISTRY
    - docker-compose -f $COMPOSE_FILE_NAME pull
    - docker-compose -f $COMPOSE_FILE_NAME -p $CI_PROJECT_NAME up -d
    ↪  --no-build
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
      when: manual
  tags:
    - tfm

# Your selected site and scanner profiles:
dast:
  stage: dast
  variables:
    DAST_AUTH_SUCCESS_IF_NO_LOGIN_FORM: "true"
  dast_configuration:
    site_profile: "pygoat"
    scanner_profile: "pygoatweb"
  when: manual

```