Investigación y evaluación de herramientas Open Source para escaneo de vulnerabilidades en el ciclo de vida de las aplicaciones, con el fin de ser integradas en los Pipelines en Azure del Grupo Éxito

David Felipe Vélez Cadavid

Trabajo de grado para optar al título de Ingeniero Electrónico

Modalidad de Práctica Cursada

Semestre de Industria

Tutores
Sebastián Isaza Ramírez, Profesor Asociado, UdeA
Juan Fernando Hincapié Zapata, Especialista SGSI Servicios TI

Universidad de Antioquia
Facultad de Ingeniería
Ingeniería Electrónica
Medellín
2024

Cita	(Vélez Cadavid, 2024)
Referencia **Estilo APA 7 (2020)**	Vélez Cadavid, D. F. (2024). *Investigación y evaluación de herramientas Open Source para escaneo de vulnerabilidades en el ciclo de vida de las aplicaciones, con el fin de ser integradas en los Pipelines en Azure del Grupo Éxito*. Semestre de Industria. Universidad de Antioquia, Medellín.

Resumen

En el ciclo de vida del desarrollo de software del Grupo Éxito, se usaba una plataforma comercial para la identificación de vulnerabilidades de seguridad en el código mediante la ejecución de pruebas SAST, SCA y DAST. Sin embargo, dada la restructuración de la dirección de TI al inicio del año 2024, se redirigió la atención hacia herramientas open source para la optimización de recursos. En este nuevo enfoque, la mayoría de los procesos de TI se debían llevar a un entorno automatizado de integración y despliegue continuo CI/CD, como parte de una estrategia para centralizar el desarrollo de software con una infraestructura definida como código en pipelines de Azure DevOps. En función de cumplir con estos requerimientos se propuso investigar e implementar herramientas open source en los pipelines de CI para reemplazar la plataforma comercial. Para ello, se hizo la evaluación de los reportes que arrojaban algunas herramientas tras la inspección de repositorios deliberadamente vulnerables, a fin de elegir las que cumplieran con los criterios de aceptación e indicadores de rendimiento favorables. Las herramientas seleccionadas permitieron cubrir pruebas de seguridad del código fuente SAST y análisis de composición de software SCA, generando el rompimiento del pipeline de forma automática ante hallazgos con severidad mayor a los umbrales definidos. En conclusión, estas herramientas generaron un impacto positivo en la compañía en relación con el ahorro de recursos, automatización de pruebas y autonomía en el control de las herramientas.

Contenido

Lista de Figuras

Lista de Tablas

1. Introducción

De acuerdo con el informe sobre amenazas en el sector retail publicado por Trustwave en 2023, la industria retail sigue siendo un objetivo importante para los ciberdelincuentes, atraídos por la gran cantidad de información de clientes que se administra, a esto se suma el avance de la inteligencia artificial que facilita la creación de correos electrónicos de phishing más persuasivos [42]. Por esta razón, el área de seguridad TI reconoce la necesidad de fortalecer las prácticas de seguridad en el ciclo de vida del desarrollo de software, garantizando que estén alineadas con la optimización de recursos y permitan proteger la información de los clientes, los activos y los sistemas de la compañía.

El desarrollo de aplicaciones inicia a partir de requerimientos asociados con la expansión del comercio electrónico y de los sistemas internos de gestión de información, tanto de clientes como de proveedores. En función de esto, se crean repositorios con sus respectivos pipelines en Azure DevOps. Sin embargo, el acompañamiento de seguridad en estos proyectos se efectúa en fases avanzadas de los entregables, lo que resulta en un aumento significativo de tiempo y recursos para la corrección de vulnerabilidades. En este estudio se pretende integrar herramientas de código abierto en las etapas tempranas del ciclo DevOps para la inspección del código. Para ello, se llevará a cabo una prueba de concepto utilizando herramientas para pruebas de seguridad de aplicaciones estáticas (SAST) y análisis de composición de software (SCA). Se evaluará el rendimiento de estas herramientas y se seleccionarán las que mejor se ajusten a las necesidades del negocio. Esta implementación generara beneficios significativos, como la realización continua y automatizada de pruebas de seguridad, mejora en la observabilidad y la eliminación del gasto derivado del uso de la plataforma comercial.

2. Objetivos

- **Objetivo general**

  Fortalecer el sistema de pruebas de seguridad en el ciclo DevSecOps de los repositorios de código de la compañía, alojados en Azure DevOps, mediante la investigación y la implementación de un piloto de pruebas para la evaluación de herramientas de aseguramiento Open Source, que puedan ser integradas en el ciclo de entregas continuas y ágiles de la gerencia de tecnología.

- **Objetivos específicos**
  - Identificar herramientas Open Source que sean adecuadas para la implementación en la verificación continua de código, con el propósito de asegurar los repositorios alojados en Azure DevOps.
  - Definir los requisitos y criterios de aceptación enfocados en la escalabilidad, rendimiento y reducción de falsos positivos, frente a los cuales se depuren y seleccionen las herramientas Open Source investigadas.
  - Implementar un piloto de pruebas con las herramientas identificadas, configurando su integración en la plantilla de un pipeline, para la evaluación de su funcionamiento en relación con el escaneo y alertamiento de vulnerabilidades en un repositorio.
  - Analizar los tableros de vulnerabilidades derivados de las pruebas realizadas, revisando los tipos de alertas y sus criticidades, con el fin de proporcionar recomendaciones pertinentes a los líderes de proyectos sobre las brechas de seguridad y sus riesgos asociados.

3. Marco Teórico

3.1. ¿Qué es DevOps?

DevOps consiste en la unión de esfuerzos del equipo de desarrollo y el de operaciones para realizar entregas de software con mayor frecuencia, apoyados en procesos estandarizados, automatización y comunicación efectiva durante todo el ciclo de desarrollo de software [1].

3.1.1. Cultura DevOps

Dada la creciente demanda de software construido a medida, con buen rendimiento y entrega continua de características, alrededor del año 2007 se empezó a hablar de la cultura DevOps, que buscaba romper esas restricciones de comunicación entre el equipo de desarrollo y el de operaciones encargado de la infraestructura. Hasta entonces desplegar código era una tarea con muchas fricciones, dada la falta de colaboración entre ambos equipos, además el modelo tradicional en cascada (Waterfall) consistía en tareas consecutivas que se completaban en varios meses e incluso años, proceso en el que las necesidades del cliente podían cambiar a pocos días de entregar el software [1]. Con la adopción de esta filosofía de trabajo el cliente estaría más enterado de los avances y para las empresas significaba ahorro de tiempo, esfuerzo y dinero, que se gastaban por correcciones en etapas avanzadas de los proyectos [2].

3.1.2. Ciclo de vida de DevOps

En la figura 1 se destacan las principales etapas del ciclo de vida de DevOps, que va desde el descubrimiento de las necesidades del usuario hasta el despliegue de un software que las soluciona. En total son 8 fases; la mitad izquierda del ciclo está compuesta por 4 fases que corresponden al equipo de desarrollo y la mitad derecha contiene las 4 fases restantes que corresponden al equipo de operaciones [1]. Las etapas se abordan como un todo, es decir, a pesar de que hay responsabilidades distribuidas se promociona la comunicación y el trabajo coordinado, estimulando una mayor calidad y velocidad en la entrega de software. El proceso está estructurado de la siguiente manera:

- **Descubrir (Discover):** Exploración de ideas.
- **Planificar (Plan):** Tableros de priorización y división de tareas.
- **Compilar (Build):** Desarrollo y compilación de código.

- **Probar (Test):** Ejecución de pruebas de calidad e integración del código.
- **Desplegar (Deploy):** Publicación de funcionalidades del software.
- **Operar (Operate):** Aseguramiento de la disponibilidad.
- **Observar (Observe):** Seguimiento de indicadores y alertamiento ante incidencias.
- **Retroalimentar (Feedback):** Revisión de los obstáculos que se presentaron para encontrar oportunidades de mejora.

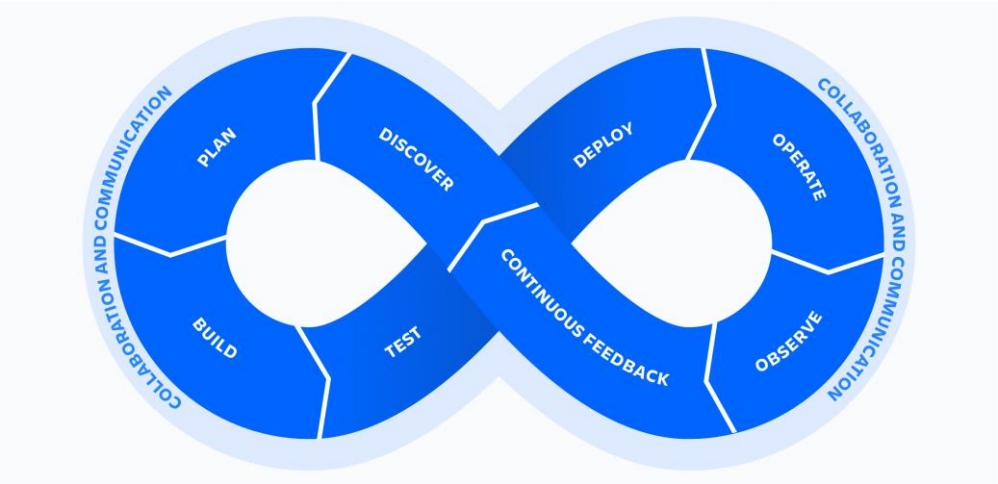


Figura 1. Ciclo de vida de DevOps [1].

3.1.3. Gestión de DevOps

La adopción de la cultura DevOps como un marco de trabajo esta intrínsicamente relacionado con los siguientes conceptos:

- **Sistema de control de versiones GIT:** Permite realizar actualizaciones y mantenimiento del código que se desarrolla a través de un historial de cambios, dando mayor soporte a las aplicaciones. Comprende opciones bien definidas de fusión de código y de recuperación de una versión específica, si se ha cometido un error de implementación. En la actualidad es un sistema muy usado por su fiabilidad y la facilidad con la que se puede trabajar colaborativamente en los equipos, para el desarrollo tanto de proyectos pequeños como a gran escala [2].
- **Plataformas de desarrollo colaborativo:** El trabajo colaborativo es parte fundamental en las empresas, cuando se trata del desarrollo de software una de las formas más productivas de trabajar es usando plataformas que brinden las herramientas necesarias

para agilizar la puesta en producción. Existen varias plataformas reconocidas con estas características como lo es Azure DevOps, GitHub, GitLab, Bitbucket, entre otras [2].

- **Flujo de trabajo**: Hace referencia a la forma en la que se organiza el entorno de trabajo en plataformas colaborativas para el desarrollo de software. Existen diversos flujos de trabajo, sin embargo, con frecuencia se usa Git Flow y Trunk Based Development [3]. El primero consiste en generar 3 ramas en un repositorio con propósitos específicos; la rama develop en la que se desarrollan funciones y se hace pruebas de garantía de calidad (QA, Quality Assurance). La rama reléase donde se verifica que las funcionalidades desarrolladas cumplen con los puntos de control de calidad (QG, Quality Gates) y se documenta el código. Finalmente, la rama main donde se concentra el código que se desplegará en producción, es decir, código que ha sido testeado y cumple con los requerimientos [4]. Por otra parte, está el flujo de trabajo Trunk Based Development que consiste en una rama principal main a partir de la que se crean ramas llamadas features para el desarrollo de funcionalidades [3].

  Git Flow es principalmente usado en organizaciones pequeñas y medianas, porque se requiere mucho más control sobre los procesos. Los equipos pueden estar conformados por programadores desde nivel junior y todas las solicitudes de cambio deben ser aprobadas mediante un Pull Request (PR) por un analista o líder de desarrollo. Trunk Based es más usado en organizaciones grandes por la necesidad de entregas mucho más frecuentes y con equipos de desarrolladores especializados donde la autogestión es uno de los factores fundamentales, se da mayor responsabilidad al desarrollador para fusionar su código sin tantos filtros, pero no significa que haya un desgobierno en los procesos [4].
- **Metodologías agiles:** DevOps no es ajeno a las metodologías agiles como lo son Scrum, Kanban y Lean, puesto que permiten una estructuración clara, bien definida y colaborativa de los proyectos [5].
- **Integración continua CI:** Consiste hacer contribuciones periódicas y generalmente de poca extensión, al código fuente alojado en un gestor de código como puede ser GitHub. De esta integración esta encargado el equipo de desarrollo, que a partir de commits y pull requests van agregando nuevas funcionalidades según los requerimientos [2].
- **Despliegue continuo CD:** Es la etapa donde se empaqueta el código y se despliega en producción, es decir, se pone a disposición del cliente o usuario final. Aquí la

intervención del equipo de operaciones es donde toma mayor importancia, porque se encargan de que la infraestructura que cubre bases de datos, plataformas de despliegue y otros recursos, funcionen adecuadamente para garantizar la disponibilidad [2].

- **Automatización:** Consiste en el desencadenamiento de acciones como compilaciones, pruebas y alertas frente a eventos relacionados con integración de código [1].
- **Microservicios:** Se trata de dividir una aplicación compleja en servicios soportados individualmente e interconectados [1].
- **Supervisión:** En DevOps se supervisa el proceso de desarrollo para responder eficientemente ante necesidades o problemas del cliente en etapas tempranas [1].

3.2. ¿Qué es Azure DevOps?

Azure DevOps de Microsoft es una plataforma dirigida al software como servicio (SaaS, Software as a Service) que opera sobre la infraestructura de computación en la nube de Azure. Está compuesta por diversas herramientas que permiten gestionar todo el ciclo de desarrollo de software, facilitando el trabajo colaborativo con Git, la implementación de metodologías ágiles y la integración y entrega continua de manera eficiente [24]. En el desarrollo de este proyecto se usaron varias herramientas, entre las que se destacan:

3.2.1. Azure Repos

Permite trabajar colaborativamente con repositorios Git creados directamente o clonados desde GitHub. Además, se pueden hacer actualizaciones al código desde un IDE o cliente GIT [23].

3.2.2. Azure Pipelines

Es un servicio para CI/CD de software, automatización de pruebas y publicación de artefactos. Los componentes claves de un pipeline de Azure DevOps se enseñan en la figura 2, y se definen así [23]:

- **Trigger (desencadenador).** Le indica al pipeline que se ejecute ante un evento como un commit.
- **Pipeline (Canalización).** Se compone de etapas en las que se agrupan trabajos específicos.

- **Agent (agente).** Es una máquina virtual con los recursos necesarios para compilar el software.
- **Stage (etapa).** En una etapa se definen trabajos dentro o fuera de un agente. Y estos a su vez contienen steps.
- **Steps (pasos).** Generalmente, contienen un conjunto de tasks y scripts que son unidades de código destinadas a la automatización de procesos, como conectarse a una API, enviar información a plataformas externas, etc.
- **Artifact (artefacto).** Paquetes o archivos publicados en la ejecución de un pipeline.

El proceso habitual para utilizar un pipeline implica tener un repositorio de código en Azure Repos y luego crear el pipeline. Los componentes del pipeline se definen en un archivo de configuración denominado azure-pipelines.yaml. Este archivo utiliza el formato YAML, un lenguaje de serialización de datos versátil, comúnmente empleado para crear archivos de configuración [25].

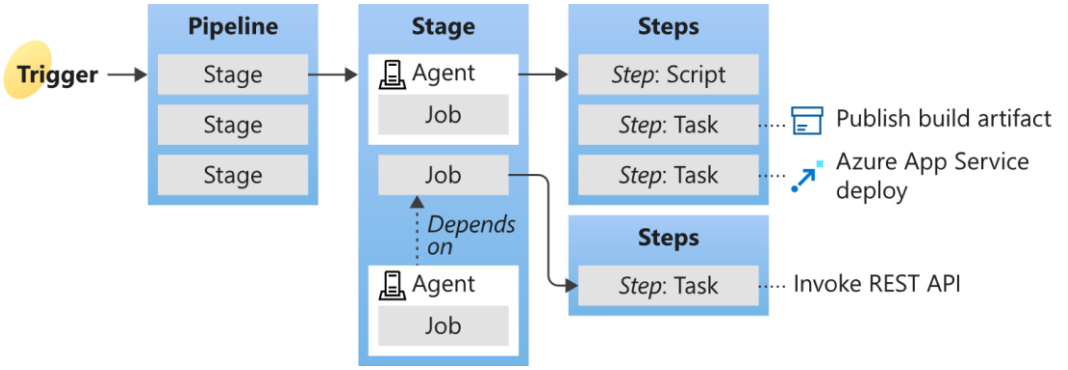


Figura 2. Componentes de un pipeline de Azure DevOps [23].

3.3. ¿Qué es DevSecOps?

DevSecOps es la evolución de la filosofía DevOps, donde se incluye la seguridad como uno de los ejes fundamentales en la integración y despliegue continuo de código. Se adapta como una necesidad y se apoya del concepto “Shift Left”, que propicia la auditoria del código en etapas tempranas del desarrollo del producto con el propósito de aceptar, rechazar o mitigar las posibles circunstancias que conllevan a vulnerabilidades. Esta filosofía garantiza mayor cohesión con los temas relacionados a seguridad porque los procesos en su mayoría pueden ser automatizados, mientras que en modelos antiguos de trabajo la seguridad se concebía como un cuello de botella por la dificulta en la implementación de las pruebas y controles.

En el ciclo de desarrollo de aplicaciones con la seguridad como un eje principal, se integran herramientas para realizar varios tipos de pruebas que generan una alerta temprana y ayudan con la corrección de problemas. Algunos de los análisis más comunes que se realizan son las pruebas de seguridad de aplicaciones estáticas (SAST), análisis de composición de software (SCA) y pruebas de seguridad de aplicaciones dinámicas (DAST) [7].

3.3.1. Gestión de DevSecOps

La gestión de DevSecOps se compone de los siguientes pilares:

- **Análisis de código**: Se realizan pruebas de seguridad del código fuente para rastrear riesgos, garantizando el cumplimiento de los requisitos de seguridad [7].
- **Administración de cambios:** Se usan herramientas para gestionar cambios de software y políticas de seguridad, evitando el reporte de vulnerabilidades por cambios imprevistos [7].
- **Administración de cumplimiento:** En este componente se garantiza que el software cumpla con las normas [7].
- **Modelado de amenazas:** Consiste en crear un plano de la estructura y los componentes de la aplicación con el fin de identificar y mitigar amenazas, tanto antes como después de que el código se ponga en producción [8].
- **Formación en seguridad:** Este ítem impulsa la capacitación de los equipos de desarrollo y operaciones en temas relacionados con buenas prácticas de seguridad [7].

3.3.2. Ciclo de vida de DevSecOps

En DevSecOps se tiene en cuenta la seguridad en cada una de las etapas del SDLC. El equipo de DevOps contribuye con la adopción de recomendaciones, buenas prácticas, monitoreo y alertamiento [10]. En la figura 3 se ilustran las seis etapas del ciclo DevSecOps: planificar, compilar, probar, desplegar, operar y monitorear. Estas etapas abarcan desde el modelado de amenazas y las pruebas de seguridad del código, hasta la recopilación de logs, la aplicación de parches y la supervisión de indicadores de alto impacto.

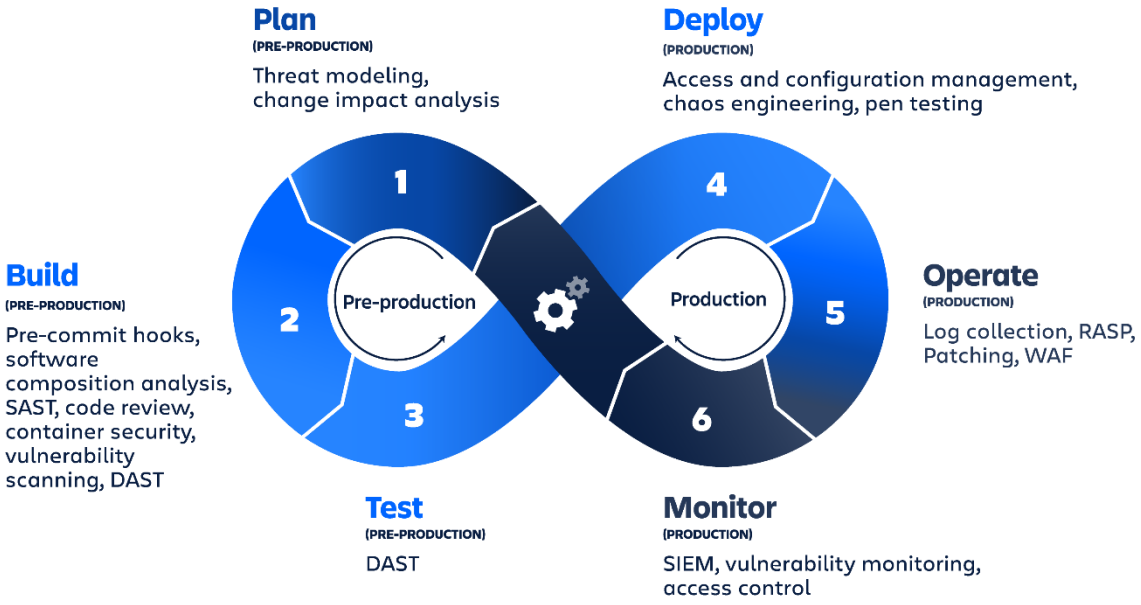


Figura 3. Ciclo de vida de DevSecOps [10].

3.3.3. Pruebas de seguridad de aplicaciones estáticas (SAST, Static Application Security Testing)

A este tipo de pruebas se les conoce como pruebas de código fuente o de caja blanca. Consisten en analizar el código en búsqueda de posibles riesgos de seguridad, habitualmente mediante la identificación de patrones. Uno de los estándares ampliamente utilizados para el etiquetado de debilidades de seguridad es CWE (Common Weakness Enumeration), gestionado por la organización MITRE, la cual además publica anualmente un ranking de las 25 debilidades más comunes [9]. Las pruebas SAST se componen de 3 tipos de alertas; alertas de seguridad, buenas prácticas y mantenimiento. En las 2 primeras se promociona el desarrollo limpio, estructurado y escalable de software. Mientras que las alertas de seguridad buscan que los riesgos derivados de configuraciones incorrectas en el código sean corregidos, evitando la exfiltración de información con técnicas como inyección de comandos XSS, inyección SQL y acceso a recursos protegidos mediante JWTs incorrectamente firmados.

3.3.4. Herramientas para SAST

Shiftleft Scan

Es una herramienta de código abierto para SAST que potencializa su operación a través de la combinación de distintos escáneres, extendiendo el análisis a código de infraestructura, todo ello sin la necesidad de un servidor remoto. Se acopla bien a los ecosistemas de CI, con la

posibilidad de interrumpir el flujo automáticamente y la generación de informes para una posterior revisión. Además, puede rastrear amenazas de seguridad en 29 lenguajes [11].

Filosofía de escaneo

- El análisis se realiza localmente, es decir, los métodos, reglas e incluso la base de datos de vulnerabilidades se descargan al momento de la ejecución de la herramienta [11].
- Implementación sencilla y sin necesidad de profundizar en conceptos complejos para efectuar el escaneo en canalizaciones CI [11].

Semgrep

Semgrep es una herramienta especializada en pruebas SAST, tiene una versión comercial y otra de código abierto (OSS, Open Source Software). Esta última es ampliamente usada en virtud de que maneja una sintaxis sencilla para escribir reglas personalizadas, a diferencia de otras herramientas que usan declaraciones de expresiones regulares o un lenguaje especifico de dominio (DSL, Domain-Specific Language) [12]. En términos de cobertura Semgrep puede detectar vulnerabilidades en 25 lenguajes de programación, a partir de patrones que identifica en el binario del código fuente [13]. Además, incluye reglas mapeadas desde otras herramientas que son usadas para el análisis de lenguajes particulares, por ejemplo, Bandit para Python y Eslint para JavaScript.

Ventajas de Semgrep OSS

- Escaneo rápido.
- Puede configurarse para realizar escaneos completos o increméntales de directorios.
- Usa metavariables para un descubrimiento semántico de más cobertura [12].
- Determinista, el resultado no cambia ante el escaneo en múltiples ocasiones de la misma entrada, es decir, es reproducible e idempotente [12].

Conjuntos de reglas de Semgrep

La elección responsable del conjunto de reglas para el escaneo es fundamental si se desea conseguir reportes con bajas tasas de falsos positivos. Cada conjunto de reglas (rulesets) tiene un propósito, a continuación, se mencionan algunos de los más conocidos:

- **Ruleset r/all:** Se aplican todos los conjuntos de reglas disponibles en la base de datos de Semgrep. Aunque el análisis puede generar avisos con alta incidencia de falsos positivos.
- **Ruleset p/default:** Este conjunto reúne las reglas más importantes de los demás conjuntos. Suele usarse para un escaneo general en cualquier tipo de ambiente y lenguaje soportado [14].
- **Ruleset p/r2c-security-audit:** Es un conjunto de reglas que están diseñadas para ejecutarse en entornos de CI, tienen mayor auditoria y bajo índice de falsos positivos. Son 226 reglas que están dirigidas a encontrar problemas de seguridad [14].
- **Ruleset p/cve-top-10:** OWASP una organización con bastante renombre en la ciberseguridad que publica anualmente el top 10 de las vulnerabilidades de mayor severidad, impacto y probabilidad de explotación. Este conjunto con 524 reglas identifica si el código tiene alguna de las vulnerabilidades del ranking anual [14].
- **Ruleset p/cwe-top-25:** Este ruleset tiene configuradas 215 reglas para verificar los riesgos del catálogo de CWEs [14].

3.3.5. Análisis de composición de software (SCA, Software Composition Analisys)

Muchas de las funcionalidades que requieren los proyectos de software, ya han sido desarrolladas y sintetizadas por terceros. En consecuencia, pueden ser usadas para acelerar la escritura de código y extender las bondades del software. Sin embargo, hay que tener prudencia al usar estas librerías prefabricadas puesto que no están exentas de ser corrompidas. En consecuencia, es imprescindible realizar análisis de composición de software (SCA) donde se verifica si las dependencias están libres de vulnerabilidades y en caso de que no, reportarlas.

El análisis de composición de software consiste en usar una herramienta que realiza todo el flujo de comprobación. Se empieza por la lectura de las dependencias del proyecto, seguidamente se realiza el cotejo contra las bases de datos oficiales y finalmente se genera un reporte con las dependencias afectadas. Allí se incluye información de versiones expuestas, si tienen parche o no, nivel de gravedad según el sistema común de puntuación de vulnerabilidades (CVSS, Common Vulnerability Scoring System) y el detalle del riesgo según el sistema de vulnerabilidades y exposiciones comunes (CVE, Common Vulnerabilities and Exposures).

3.3.6. Herramientas para SCA

Existen herramientas para análisis de composición de software comerciales como Snyk y Veracode. Sin embargo, en este trabajo son de interés las herramientas open source que se tratan más al detalle a continuación.

Dependency Check de OWASP

Esta herramienta admite 12 lenguajes, incluyendo proyectos desarrollados en C#, Java, Python y JavaScript [12].

Métodos de escaneo

Las principales técnicas que se usan para el escaneo de paquetes son; análisis de archivos xml con expresiones XPath, expresiones regulares para inspección de texto y rastreo de metadatos. Esta herramienta no usa la técnica de verificación de hashes sino una técnica de verificación de metadatos para reconocer la dependencia, es decir, recopila información de enumeración de plataformas comunes (CPE, Common Platform Enumeration), para luego realizar la búsqueda en bases de datos oficiales de CVEs, a partir de los datos recopilados llamados Evidencia, que contienen registros particulares como la versión, el cliente y el producto [16]. Mantener una basta base de datos de hashes puede ser complejo y también es susceptible a errores, debido a que los paquetes o librerías pueden variar mínimamente si el usuario compila la dependencia en el código fuente en lugar de referenciarla en un archivo de configuración.

Bases de datos

Dependency usa la base de datos nacional de vulnerabilidades (NVD, National Vulnerability Database) gestionada por Instituto Nacional de Estándares y Tecnología (NIST, National Institute of Standards and Technology). Otras bases populares y usadas por Dependency Check son [17]:

- **OSS Index.** Contiene información acerca de vulnerabilidades conocidas en dependencias de código abierto y es soportada por la empresa Sonatype.
- **RetireJs.** Base de datos que recopila vulnerabilidades en dependencias de JavaScript.
- **Npm Audit.** Contiene registros de vulnerabilidades presentes en paquetes Node.js.
- **Bundler-audit.** Útil para comprobar vulnerabilidades en paquetes Ruby.
- **VFeed.** Repositorio que contiene metadatos relacionados con vulnerabilidades adquiridas de diferentes fuentes.

- **Victims CVE Database**. Hace parte del proyecto The Victims Project y recopila metadatos relacionados con vulnerabilidades en lenguajes como Python y Java.

OSV – Scanner de Google

En un principio Google creo la base de datos OSV.dev, con el objetivo de centralizar las dependencias de código abierto con vulnerabilidades conocidas. En consecuencia, se construyó OSV-Schema que consiste en un formato simple a través del cual otras bases de datos pueden exportar su catálogo [18]. Posteriormente, se lanzó OSV-Scanner que usa la información de OSV.dev por medio de su API.

Bases de datos

OSV cuenta con su propia base de datos llamada OSV.dev, que es open source y distribuida, además recopila información de vulnerabilidades de múltiples fuentes oficiales como [19]:

- **NVD.** Base de datos nacional de vulnerabilidades.
- **Github Security Advisores.** Repositorio de Github donde se concentra una extensa lista de alertas de seguridad.
- **Snyk**. Una plataforma con su propia base de datos.

Infraestructura de la base de datos de OSV – Scanner

La infraestructura en términos generales de la base de datos OSV.dev se puede observar en la figura 4, donde se incorporan otras bases que admiten el formato OpenSSF [21]. En este proceso se aplica la bisección y análisis de versiones para identificar con precisión las variantes afectadas. Los usuarios pueden usar la base de datos a través de la API o simplemente usando el escáner [20].

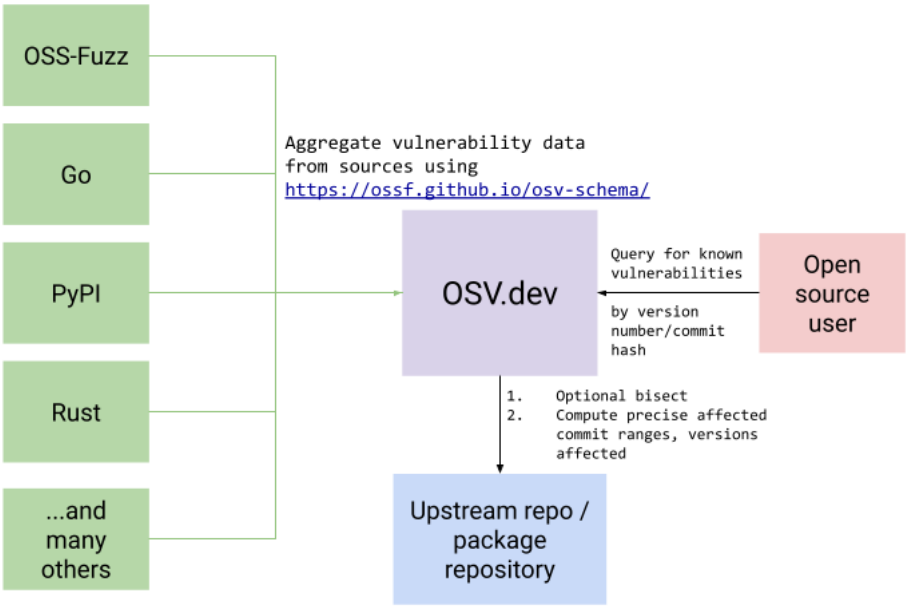


Figura 4. Infraestructura general de la base de datos OSV.dev [21].

Métodos de identificación

OSV realiza el análisis de archivos de configuración, validación de hashes e identificación de metadatos. También construye un árbol completo de dependencias incluidas las transitorias, es decir, dependencias que se generar a partir de otras [22].

Ventajas

- Alta disponibilidad.
- Bases de datos actualizadas en el 99% del tiempo con obsolescencia de solo 15 minutos.
- La API no presenta limitaciones en la tasa de uso.
- Velocidad en la consulta y respuesta.
- Identificación precisa [22].

Trivy de Aqua Security

Trivy es un escáner de seguridad open source potente y versátil con soporte para más de 13 lenguajes [32]. Tiene la capacidad de escanear dependencias, configuraciones incorrectas de IAC, secretos y licencias de software. En cuanto a dependencias examina paquetes de sistema operativo, paquetes específicos de lenguajes y componentes de Kurbenetes [26].

Método de escaneo

Para las dependencias, inspecciona las versiones y las contrasta con bases de datos de vulnerabilidades conocidas [26]. En relación con configuraciones incorrectas de IAC, posee reglas de verificación predefinidas y permite definir reglas personalizadas [28]. En términos de secretos, escanea contraseñas, tokens y claves de API utilizando reglas integradas [29]. Además, se escanean imágenes de contenedores para chequear licencias y generar avisos según la Clasificación de Licencias de Google (Google License Classification) [31], [33].

Fuentes de datos

Consta de 2 tipos de bases de datos que se distribuyen a través del Registro de Contenedores de GitHub (GHCR, GitHub Container Registry) así [26]:

- **Base de datos de vulnerabilidades**: Una base de datos con vulnerabilidades que se actualiza en Github cada 6 horas. Contiene información de Red Hat, NVD, Debian, etc.
- **Base de datos de índices Java**: Una base de datos que solo se descarga cuando se escanean archivos de configuración JAVA. Esta se actualiza una vez al día en Github.

El consumo de avisos de seguridad en Trivy, en relación con los sistemas operativos, prioriza las fuentes de los proveedores. Esto es importante porque los proveedores pueden aplicar correcciones al software en versiones anteriores a las que se publican como parchadas. De este modo, se evitan falsos positivos [26].

Por otra parte, la elección de la severidad de una exposición se realiza priorizando los reportes de los proveedores. Esto se debe a que la NVD no conoce cómo se distribuye el software, lo que puede causar discrepancias en los puntajes de severidad. Por ejemplo, mientras que la NVD puede calificar una vulnerabilidad como alta, el proveedor Red Hat podría asignarle un puntaje bajo. El grado de severidad se asigna según el Sistema Común de Puntuación de Vulnerabilidades (CVSS) [27].

Opciones de configuración

Trivy es ampliamente configurable. Entre sus opciones más relevantes se encuentran [34]:

- **--ignore-unfixed.** Para evitar alertas sobre vulnerabilidades que no tienen parche se declara true.

- **--scanners.** Permite especificar los tipos de problemas a buscar como vuln, misconfigurations, secret y lincense. Se pueden habilitar las 4 opciones separadas por coma.
- **--exit-code 1.** Genera código de error 1 ante una severidad.
- **--severity.** Filtra el aviso por gravedad ya sea critical, high, medium, low o unknown.

4. Metodología

4.1. Investigación de herramientas open source

Se investigaron un conjunto de herramientas open source para la realización de pruebas de seguridad SAST y SCA. Herramientas de vanguardia, populares, confiables y apropiadas para implementar en entornos de integración continua (CI).

En un primer acercamiento con este tipo de software open source, se pretendía conseguir la mayor cobertura posible de pruebas de seguridad con una sola herramienta. Sin embargo, en el mundo del código abierto, es complicado encontrar herramientas que aborden varios tipos de pruebas con un buen grado de precisión, por la necesidad de realizar actualizaciones, documentación y soporte, muy frecuentemente. En consecuencia, se debía elegir una herramienta para análisis SAST y otra para SCA.

Para hallar herramientas conforme a las necesidades de la compañía, se realizó una exploración en revistas relacionadas con ciberseguridad, conferencias y repositorios populares de Github. Como resultado de la exploración se preseleccionaron las siguientes herramientas.

Para SAST:

- Scan de Shiftleft.
- Semgrep OSS de r2c.

Para SCA:

- Dependency Check de OWASP.
- OSV Scanner de Google.
- Trivy de AquaSecurity.

En la figura 5 se ilustra el nivel de aceptación, medido en la cantidad de estrellas adquiridas a lo largo del tiempo, de las herramientas preseleccionadas para SAST. En cuanto a Semgrep, se observa un crecimiento constante entre 2021 y 2024, mientras que el crecimiento de Scan no fue considerable en este período.

La aceptación de las herramientas preseleccionadas para SCA se muestra en la figura 6. Respecto a Dependency Check, se observa una mayor madurez en el mercado respecto a OSV y Trivy. Además, su popularidad fue aumentando en el transcurso de los años. Sin embargo, Trivy la superó en menos de un año. De forma similar, OSV igualó su popularidad en aproximadamente el mismo período.

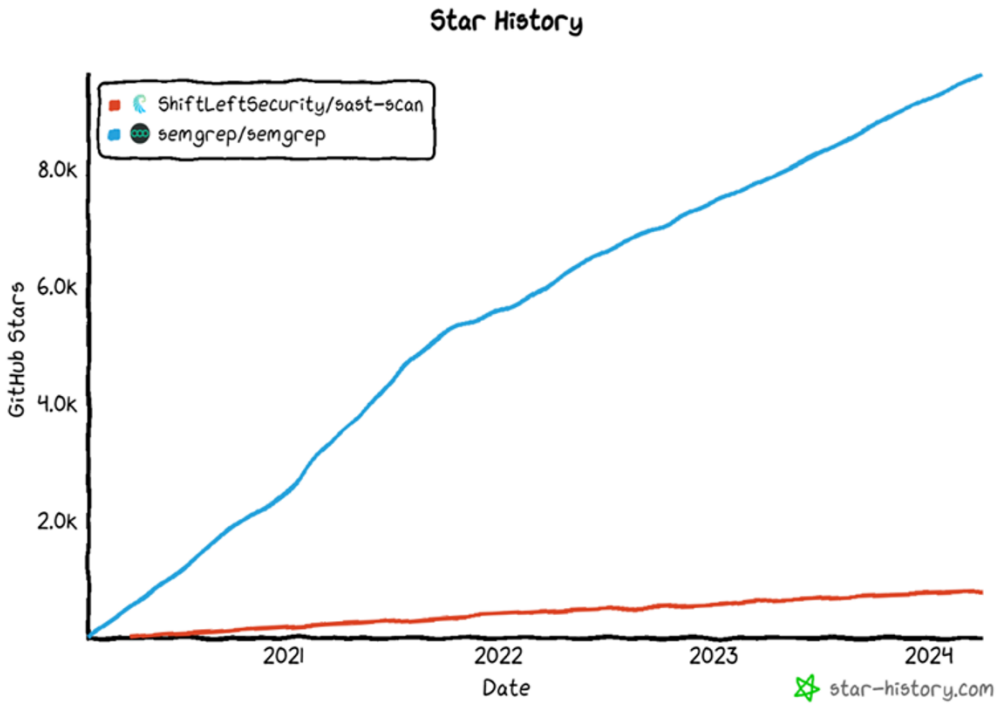


Figura 5. Popularidad de Scan y Semgrep [35].

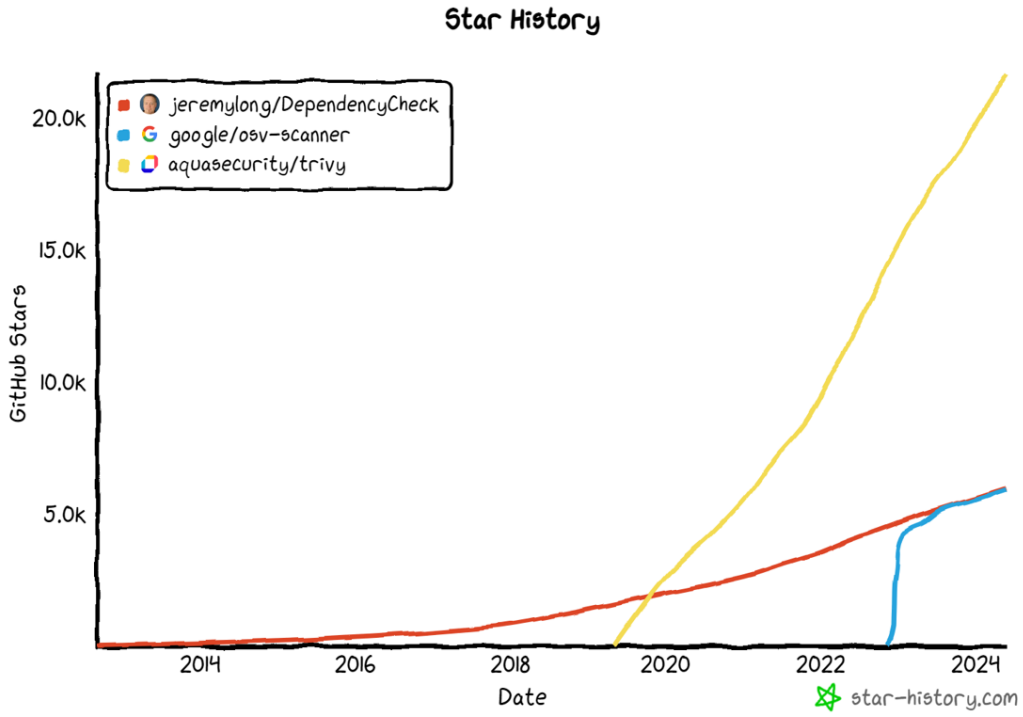


Figura 6. Popularidad de Dependency, OSV y Trivy [36].

4.2. Requisitos de aceptación

Para determinar la afinidad de las herramientas con los ambientes y tecnologías de desarrollo de la dirección de TI, se realizaron tablas de requisitos funcionales y no funcionales tanto para las herramientas SAST y SCA. En cada requisito se priorizo un criterio de aceptación, asignándole un puntaje con la siguiente correspondencia:

- El Puntaje (P) es 1 si le herramienta no cumple con el criterio de aceptación.
- El Puntaje es 2 si la herramienta cumple parcialmente.
- El Puntaje es 3 si la herramienta cumple.

La interpretación de los resultados se realizó a partir del porcentaje de Afinidad, que consiste en la sumatoria de los puntajes obtenidos entre el producto del máximo puntaje posible y el número de casos, como se muestra en la siguiente ecuación.

$$Afinidad\% = \frac{\sum_{i=1}^{N} P_i}{3N} \times 100$$

Donde Pi hace referencia al puntaje obtenido para cada caso evaluado de los requisitos funcionales y no funcionales. El 3 es el puntaje máximo que se puede obtener y N es el total de criterios evaluados.

En la tabla 1 se especifican los requisitos funcionales, mientras que en la tabla 2 se exponen los requisitos no funcionales que se utilizaron para evaluar Scan y Semgrep. La afinidad de Scan con los criterios de aceptación fue alta con un 75,76%, pero presento deficiencias importantes en la documentación, la personalización de las opciones de escaneo y el soporte. No obstante, Semgrep cumplió al pie de la letra con la mayoría de los criterios, alcanzando un porcentaje de afinidad del 96,97% con ventajas destacadas como actualizaciones frecuentes, comunidad activa y facilidad para usar reglas personalizadas.

Tabla 1. Requisitos funcionales para herramientas SAST.

Requisitos funcionales		**Herramientas**	
Requisito	**Criterio de aceptación**	**Scan**	**Semgrep**
		P	**P**
Integración en pipelines de Azure DevOps	Se puede instalar con binario o imagen de Docker	3	3
Detección de vulnerabilidades en múltiples lenguajes	Analiza lenguajes como C#, Java y Python	3	3
Opciones de configuración	Se pueden configurar parámetros de escaneo y usar reglas con baja tasa de falsos positivos	2	3
Generación de reportes	Genera reportes en formato json, html o sarif	3	3
Puntaje total		11	12

Tabla 2. Requisitos no funcionales para herramientas SAST.

Requisitos no funcionales		**Herramientas**	
Requisito	**Criterio de aceptación**	**Scan**	**Semgrep**
		P	**P**
Licenciamiento	Tiene licencia como Apache 2.0 o LGPL 2.1	3	3
Rendimiento y escalabilidad	Escanea rápido y recursivamente	2	3
Personalización de reglas para el escaneo	Ejecuta reglas personalizadas	1	3
Bases de datos públicas y oficiales	Consulta bases de datos como la NVD, OSV.dev o Github Advisory	3	3
Documentación	Documentación amplia y precisa	1	2
La herramienta tiene soporte activo	Mejora constante por la empresa y la comunidad	1	3
Alineación con principales estándares de seguridad	Especifica el CWE relacionado con la vulnerabilidad	3	3
Puntaje total		14	20
Afinidad del total de requisitos [%]		75,76	96,97

En la tabla 3 se especifican los requisitos funcionales, mientras que en la tabla 4 se exponen los requisitos no funcionales que se utilizaron para evaluar Dependency Check, OSV y Trivy. El porcentaje de afinidad reveló que Trivy cumplió en su totalidad con los requerimientos. Por otro lado, OSV tuvo un 12% de desacierto, debido a la falta de incorporación de plantillas para presentar los hallazgos más legibles y a que algunas de las opciones de escaneo relevantes estaban en fase experimental. Por otro lado, Dependency presentó un 18% de desacierto, vinculado a los altos tiempos de escaneo y a los reportes poco cómodos de leer.

Tabla 3. Requisitos funcionales para herramientas SCA.

Requisitos funcionales		**Herramienta**		
Requisito	**Criterio de aceptación**	**Dependency**	**OSV**	**Trivy**
		P	**P**	**P**
Integración en pipelines de Azure DevOps	Se puede instalar con binario o imagen de Docker	3	3	3
Detección de vulnerabilidades en múltiples lenguajes	Analiza lenguajes como C#, Java y Python	3	3	3
Uso de plantillas	Permite usar plantillas para generar reportes más legibles	1	1	3
Opciones de configuración	Se pueden configurar los parámetros de escaneo	2	2	3
Generación de reportes	Reportes en formato json, html, xml o extensión de Azure DevOps	2	2	3
Puntaje total		11	11	15

Tabla 4. Requisitos no funcionales para herramientas SCA.

Requisitos no funcionales		**Herramienta**		
Requisito	**Criterio de aceptación**	**Dependency**	**OSV**	**Trivy**
		P	**P**	**P**
Licenciamiento	Tiene licencia Apache 2.0 o BSD	3	3	3
Rendimiento y escalabilidad	Escaneo rápido y recursivo	1	3	3
Bases de datos públicas y oficiales	Consulta bases de datos como la NVD, OSV.dev o Github Advisory	3	3	3
Documentación	Tiene documentación detallada	3	3	3
Alineación con principales estándares de seguridad	Especifica el CVE relacionado con la vulnerabilidad	3	3	3
Versiones de las dependencias afectadas y parches	Informa sobre versiones afectadas y parches	3	3	3
Puntaje total		16	18	18
Afinidad del total de requisitos [%]		81,82	87,88	100,00

4.3. Análisis del rendimiento de las herramientas preseleccionadas

En una cuenta de Azure DevOps creada con una suscripción gratuita, se realizó un laboratorio de pruebas con el propósito de evaluar el rendimiento de las aplicaciones preseleccionadas. Allí se clonaron los repositorios y se crearon los respectivos pipelines para 2 aplicaciones web vulnerables. Benchmark de OWASP escrita en Java para examinar las 2 herramientas SAST y PyGoat escrita en Python para examinar las 3 herramientas SCA [37], [38]. Las tareas que se definieron y los parámetros para cada herramienta se enuncian a continuación.

4.3.1. Configuración de Scan

La figura 7 corresponde a la configuración de la tarea para la herramienta Scan. Debido a que en la tarea se usa Docker, se montó un volumen del código fuente en app y un volumen para almacenar los reportes en reports. De las opciones propias de Scan se usó –src para señalar el directorio a escanear y con –out_dir el directorio donde se almacenan los reportes.

```
- task: Bash@3
  displayName: "Ejecuntando Scan"
  inputs:
    targetType: 'inline'
    script: |
      docker run \
      -v "$(Build.SourcesDirectory):/app:cached" \
      -v "$(Build.ArtifactStagingDirectory):/reports:cached" \
      shiftleft/sast-scan:latest scan \
      --src /app \
      --out_dir /reports
  continueOnError: false
```

Figura 7. Tarea para Scan.

4.3.2. Configuración de Semgrep

En la configuración de Semgrep que se muestra en la figura 8 se contemplaron los siguientes ítems:

- Aplicar conjuntos de reglas que tuvieran un proceso de auditoría y testeo maduro para reducir la probabilidad de obtener falsos positivos con el parametro --config.
- Desactivar él envió de métricas o información relacionada con el código a la compañía desarrolladora de la herramienta, con el parámetro –metrics=off.
- Omitir verificaciones de versión comercial con el parámetro –pro.

- Ocultar mensajes de los procesos internos, para producir una salida más limpia de los hallazgos en el Log del pipeline con -q (quiet, silencioso).
- Especificar el directorio a escanear con /src y el nombre del archivo para el reporte de las vulnerabilidades con –json-output.

```
- task: Bash@3
  displayName: "Ejecutando Semgrep"
  inputs:
    targetType: 'inline'
    script: |
      docker run \
      -v "$(Build.SourcesDirectory):/src:cached" \
      -v "$(Build.ArtifactStagingDirectory):/reports:cached" \
      semgrep/semgrep:latest \
      semgrep scan \
      --config "p/default" \
      --config "p/r2c-security-audit" \
      --config "p/owasp-top-ten" \
      --config "p/cwe-top-25" \
      --metrics=off \
      --pro \
      -q \
      /src \
      --json-output /reports/reporte-semgrep.json
```

Figura 8. Tarea para Semgrep.

4.3.3. Configuración de Dependency Check

Similarmente a las configuraciones anteriores también se definió src para el directorio fuente con --scan y el directorio de reportes con –out como se ilustra en la figura 9. Aunque se añadieron otras opciones como:

- Formato del reporte en HTML con –format.
- Nombre del proyecto con –project.
- Puntaje de ruptura del pipeline a partir de un CVSS mayor a 7.0 con –failOnCVSS

```
- task: Bash@3
  displayName: 'Ejecutando DependencyCheck'
  inputs:
    targetType: 'inline'
    script: |
      docker run \
      -v "$(Build.SourcesDirectory):/src:cached" \
      -v "$(Build.ArtifactStagingDirectory):/reports:cached" \
      owasp/dependency-check:9.2.0 \
      --scan /src \
      --format "HTML" \
      --project "OWASP Dependency Check" \
      --out /report \
      --failOnCVSS 7.0
  continueOnError: false
```

Figura 9. Tarea para Dependency Check.

4.3.4. Configuración de Trivy

Puesto que Trivy posee una extensión que se puede instalar desde el Marketplace de Azure DevOps la configuración de las opciones se realizó en input options como se puede apreciar en la figura 10. A continuación, se detallan.

- **--scanners.** Permite establecer qué tipo de análisis se realiza, en este caso búsqueda de vulnerabilidades, misconfigurations (configuraciones incorrectas), secret (secretos) y license (licencias).
- **--exit-code 1**. Depende directamente de la definición del parámetro severity para generar un código de salida 1, que provoca la ruptura del pipeline.
- **--severity.** Solo mostrar hallazgos con severidad crítica y alta.
- **--ignorefile.** Se instancia un archivo para omitir vulnerabilidades aceptadas por su naturaleza. Esto se explica más adelante.

```
- task: trivy@1
  displayName: 'Ejecutando Trivy'
  inputs:
    options: |
      --scanners "vuln,misconfig,secret,license"
      --exit-code 1
      --severity "CRITICAL,HIGH"
      --ignorefile .trivyignore
    version: latest
    ignoreUnfixed: true
    path: .
  continueOnError: false
```

Figura 10. Tarea para Trivy.

4.3.5. Configuración para OSV Scanner

Finalmente, para la tarea de OSV que se ilustra en la figura 11 se activó los detalles de ejecución en el Log con –verbosity, la generación de reportes en formato json con –format y el escaneo recursivo con -r.

```yaml
- task: Bash@3
  displayName: 'Ejecutando OSV - Scanner'
  inputs:
    targetType: 'inline'
    script: |
      docker run \
      -v "$(Build.SourcesDirectory):/src:cached" \
      -v "$(Build.ArtifactStagingDirectory):/reports:cached" \
      ghcr.io/google/osv-scanner \
      --verbosity verbose \
      --format json --output /reports/reporte-osv-scanner.json \
      -r /src
```

Figura 11. Tarea para OSV Scanner.

4.3.6. Ejecución de herramientas preseleccionadas

Adicionalmente a las configuraciones de la sección anterior, se realizaron ajustes para que el pipeline fallara ante exposiciones con gravedad crítica o alta. Como Semgrep y OSV no ofrecen reportes en formato HTML, se implementó un script para mapear los resultados que entregaban en JSON a HTML. Véase el **Anexo A** y el **Anexo B** para conocer los códigos completos.

En este apartado se muestran los reportes de las ejecuciones. Sin embargo, no se entra en detalles, ya que el análisis de resultados se aborda en la siguiente sección.

Ejecución de Scan y Semgrep para SAST.

La figura 12 muestra el resumen de la ejecución de Scan y Semgrep. Donde se aprecia que ambas herramientas rompieron el pipeline puesto que encontraron exposiciones de severidad crítica o alta.

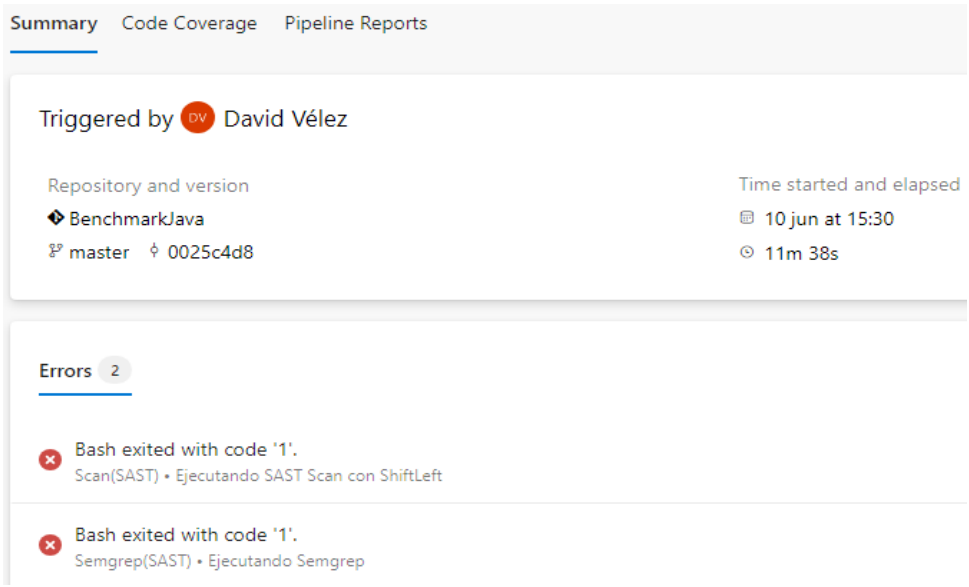


Figura 12. Resumen de ejecución de SAST.

En los resultados de ejecución de Scan que se muestran en la Figura 13, se generó un código de salida 1 que rompió el pipeline debido a que se encontraron vulnerabilidades críticas con el analizador de clases de archivos Java.

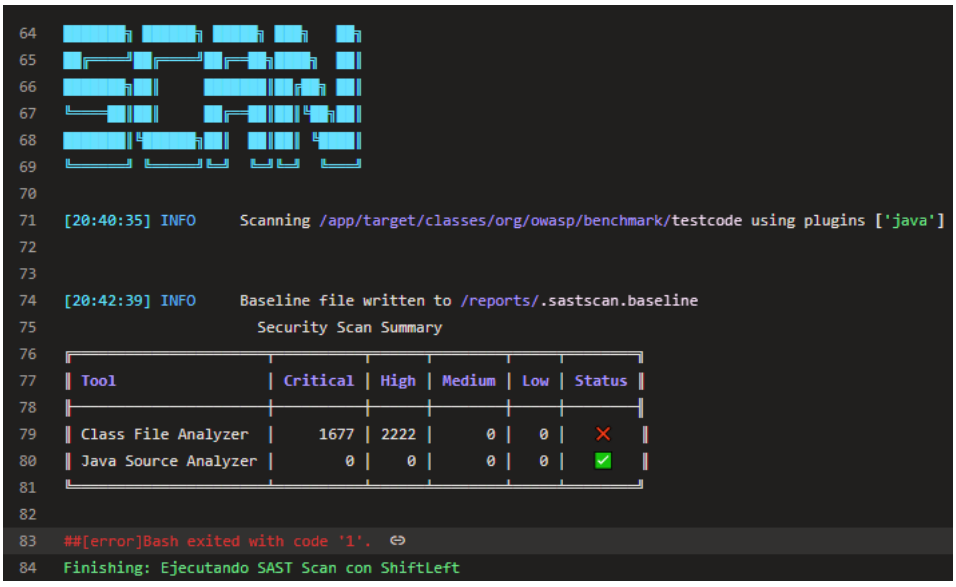


Figura 13. Reporte de Scan.

En la Figura 14 se muestran los resultados de Semgrep en una pestaña de la interfaz de Azure Pipelines, gracias a la instalación de una extensión del Marketplace llamada Html Report Viewer, que incluye una tarea predefinida para el pipeline que publica el HTML generado [40]. Esto se hizo con el propósito de facilitar la gestión de los hallazgos, ya que de lo contrario sería necesario descargar el artefacto html, abrirlo e inspeccionarlo manualmente.

El reporte brinda un nivel de detalle profundo que agiliza el proceso de remediación, a partir de la revisión de la severidad, el impacto, la probabilidad de explotación y la ubicación del CWE.

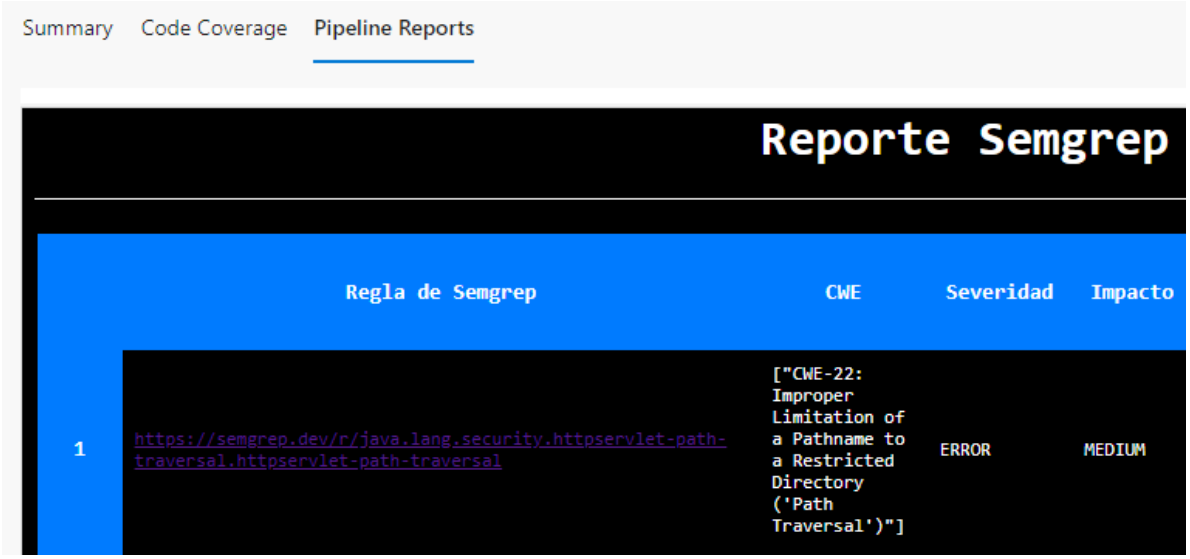


Figura 14. Reporte de Semgrep.

Ejecución de Dependency, OSV y Trivy para SCA.

El resumen de los resultados para SCA se presenta en la Imagen 15. Las tres herramientas rompieron el pipeline debido a la gravedad de las exposiciones.

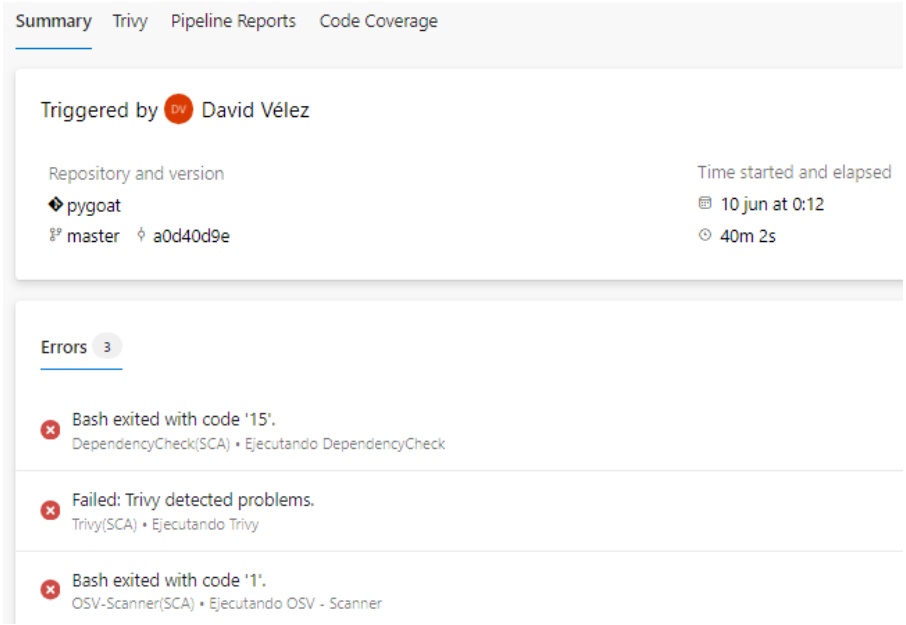


Figura 15. Resumen de ejecución SCA.

El reporte de Dependency se puede observar en la Figura 16, con información relevante como vulnerabilidades encontradas, CWE y CVSS para cada vulnerabilidad.

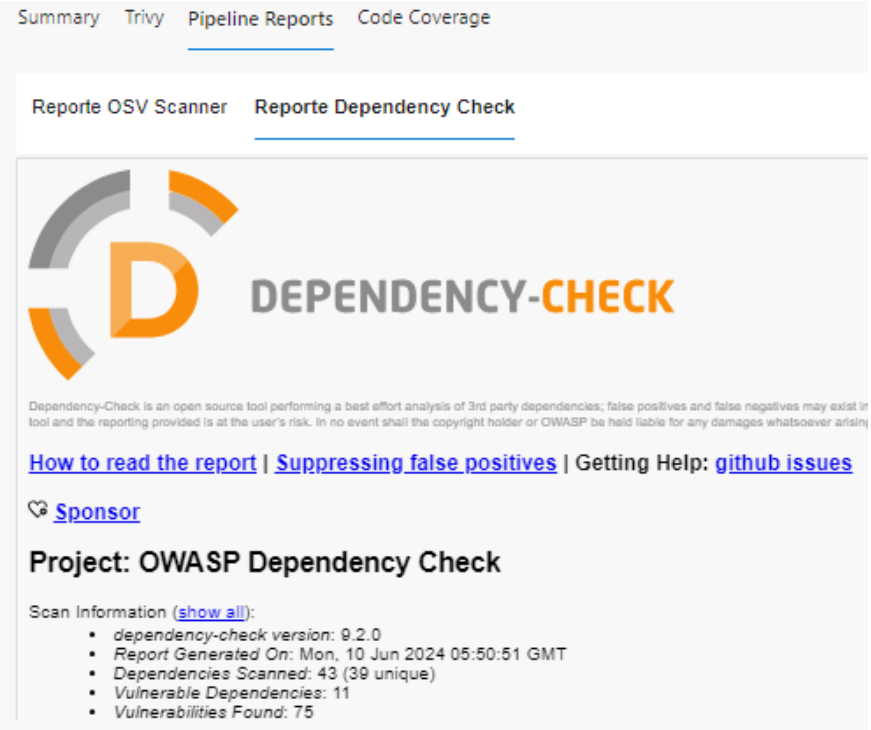


Figura 16. Reporte de Dependency.

El reporte de OSV se puede observar en la Figura 17, con información relevante para cada vulnerabilidad como CVE, severidad, CVSS, versión afectada, versión reparada y ubicación del archivo con la vulnerabilidad.

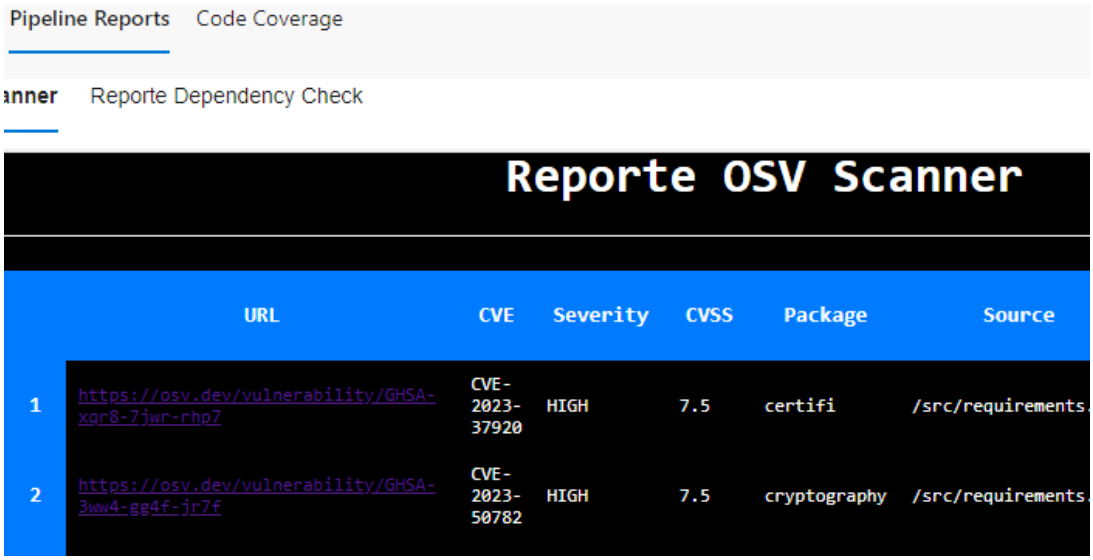


Figura 17. Reporte de OSV.

En el caso de Trivy, se instaló y usó la extensión que Aqua Security pone a disposición en el Marketplace con el objetivo de visualizar los resultados de manera más amigable [41], como se muestra en la Figura 18. No solo se encontraron problemas de dependencias, sino también

secretos e inconvenientes de infraestructura como configuraciones incorrectas de archivos para la puesta en marcha de imágenes de contenedores.

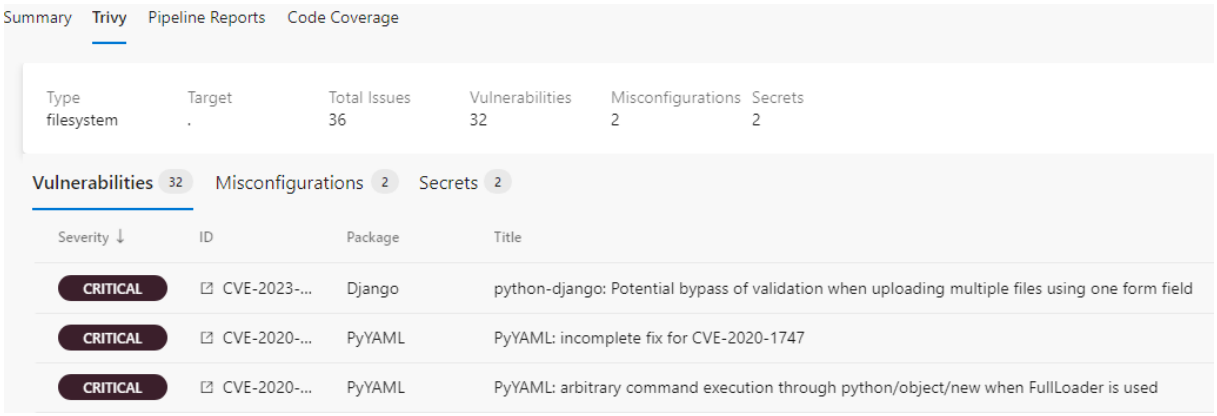


Figura 18. Reporte de Trivy.

4.3.7. Métricas de rendimiento

Una vez que las herramientas fueron parametrizadas, se realizaron las pruebas de seguridad. Tras finalizar los escaneos, se crearon indicadores sobre verdaderos positivos, falsos positivos, tiempo de ejecución, entre otros. Estos indicadores se usaron para hacer una comparativa de rendimiento, siguiendo la metodología propuesta por OWASP [39]. A continuación, se presentan los indicadores utilizados:

- Verdadero Positivo (VP) -Vulnerabilidades reales identificadas.
- Falso Negativo (FN) - Vulnerabilidades reales que no fueron detectadas.
- Falso Positivo (FP) – Cantidad de casos en los que se localizó una vulnerabilidad inexistente.
- Verdadero Negativo (VN) – Cantidad de casos identificados satisfactoriamente donde no existía vulnerabilidad.
- P - Total de vulnerabilidades presentes en una aplicación.

$$P = VP + FN$$

- N - Total de casos sin vulnerabilidad en la aplicación.

$$N = FP + VN$$

- TOTAL - Cantidad de casos analizados.

$$TOTAL = P + N$$

- Precisión (Precision, PRE) - Expresa la proporción de verdaderos positivos en relación con el total de positivos notificados (incluyendo FP) por la herramienta.

$$PRE = TP/(TP + FP)$$

- Sensibilidad (Sensibility o Recall, SEN) - Indica la proporción de verdaderos positivos en relación con el total de vulnerabilidades reales reportadas por la herramienta.

$$SEN = TP/ (TP + FN)$$

- Exactitud (Accuracy, ACC) - Denota la porción de evaluaciones correctas del total evaluaciones realizadas.

$$ACC = (VP + VN) / (VP + VN + FP + FN)$$

- Tasa de verdaderos positivos (VP %) - Fracción de verdaderos positivos del total de evaluaciones.

$$VP\% = VP/P$$

- Tasa de falsos negativos (FN %) - Fracción de falsos negativos del total de evaluaciones.

$$FN\% = FN/P$$

- Tasa de falsos positivos (FP %) - Fracción de falsos positivos del total de evaluaciones.

$$FP\% = FP/N$$

- Tasa de verdaderos negativos (VN %) - Fracción de verdaderos negativos del total de evaluaciones.

$$VN\% = VN/N$$

4.3.8. Resultados de SAST sobre la aplicación Java

La tabla 5 presenta un análisis detallado de la capacidad de las herramientas, en donde se evaluaron métricas importantes como la precisión, la sensibilidad y exactitud de cada herramienta para tener una visión general de sus fortalezas y debilidades. Todos los indicadores que se ilustran están descritos en la sección anterior.

En relación con los verdaderos positivos (VP) **Semgrep se destacó** con 1161 casos reales de 1415 posibles. En cuanto a la sensibilidad (SEN) el 97,81 % de los casos reales de vulnerabilidades fueron reportados, esto refleja que la herramienta tiene una tasa de aciertos bastante buena. En el caso de Scan los resultados no fueron muy alentadores, puesto que se presentó un porcentaje (210,87 %) muy alto de falsos positivos, y métricas como la precisión (PRE) y la exactitud (ACC) no alcanzaron ni el 30%, lo que genera desconfianza en los resultados que la herramienta entrega, lo anterior puede deberse en parte a las restricciones de configuración de los parámetros ocasionando limitantes en el funcionamiento de la herramienta,

además la gestión y revisión de tantos falsos positivos puede comprometer recursos y tiempo innecesariamente.

Tabla 5. Indicadores de rendimiento para las herramientas SAST.

	Semgrep	Scan
VP	1161	889
FN	26	526
FP	697	2794
VN	15	0
P	1415	
N	1325	
TOTAL	2740	
PRE [%]	62,49	24,14
SEN [%]	97,81	62,83
ACC [%]	61,93	21,12
VP [%]	82,05	62,83
FN [%]	1,84	37,17
FP [%]	52,60	210,87
VN [%]	1,13	0,00

Para obtener los valores de VP, FN, FP y VN se realizó un script en Python con el que se cotejaron los resultados arrojados por las herramientas SAST con la información que pone a disposición OWASP acerca de las vulnerabilidades de la aplicación Benchmark. Parte del código se puede observar en la figura 19. No obstante, el código completo se puede encontrar en el **anexo C.**

```
verdadero_positivo = 0
verdadero_negativo= 0
falso_negativo  = 0
falso_positivo = 0

for x, y in df_cwes_archivos:
    for i in df_benchmark.iterrows():
        if (i[1]['test name'] == x and (i[1]['cwe'] != 'CWE-327')):
            lista_cwes = list(y['CWE'])
            if (i[1]['cwe'] in lista_cwes) and i[1]['real vulnerability'] == True:
                verdadero_positivo += 1
                longitud_cwes = len(lista_cwes) - 1
                falso_positivo += longitud_cwes
            elif (i[1]['cwe'] in lista_cwes) and i[1]['real vulnerability'] == False:
                falso_positivo += len(lista_cwes)
            elif (i[1]['cwe'] not in lista_cwes) and i[1]['real vulnerability'] == True:
                falso_negativo += 1
                falso_positivo += len(lista_cwes)
            elif (i[1]['cwe'] not in lista_cwes) and i[1]['real vulnerability'] == False:
                verdadero_negativo += 1
                falso_positivo += len(lista_cwes)
            break
```

Figura 19. Script de Python para el cálculo de indicadores.

4.3.9. Resultados de SCA sobre la aplicación Python

La tabla 6 proporciona información al detalle de las ventajas y desventajas de Dependency, Trivy y OSV. Respecto a VP Trivy y OSV detectaron el total de vulnerabilidades reales que eran 32. Sin embargo, Dependency no reporto 6 casos reales y en su lugar tuvo un reporte de falsos positivos muy alto con una tasa de FP del 98%. Esto implica una tasa más alta de FP que la de VP con un 81,25% y consecuentemente valores de precisión (PRE) y exactitud que no alcanzan el 35%, a pesar de que la sensibilidad (SEN) estuvo sobre el 80% el tratamiento de tantos falsos positivos implica esfuerzo y gasto de recursos para su investigación.

Trivy se posiciona con una alternativa muy atractiva con 0 falsos positivos contra 1 falso positivo reportado por OSV. Aunque las métricas de ambas herramientas fueron buenas Trivy alcanzó tasas de 100 en exactitud, sensibilidad y precisión. También reportó misconfigurations y secrets que no se tuvieron en cuenta para el análisis, pero dan un valor agregado a la herramienta.

Tabla 6. Indicadores de rendimiento para las herramientas SCA.

	Dependency	**Trivy**	**OSV**
VP	26	32	32
FN	6	0	0
FP	49	0	1
VN	1	50	49
P		32	
N		50	
TOTAL		82	
PRE [%]	34,67	100,00	96,97
SEN [%]	81,25	100,00	100,00
ACC [%]	32,93	100,00	98,78
VP [%]	81,25	100,00	100,00
FN [%]	18,75	0,00	0,00

FP [%]	98,00	0,00	2,00
VN [%]	2,00	100,00	98,00

La tabla 7 manifiesta los tiempos que cada herramienta se tomó para realizar el escrutinio de las aplicaciones web. Para el caso de SAST las herramientas demoraron varios minutos por la robustes del repositorio Benchmark, sin embargo, no superaron los 7 minutos(420s). Se posiciono Scan como la más rápida con un tiempo menor a 5 minutos (300s). En términos de SCA, el diagnostico tardo menos de 1 minuto (60s) para Trivy y OSV, esto en función de que se escanean únicamente archivos de configuración. Una situación diferente ocurrió para Dependency Check que tardo alrededor de 38 minutos (2297s), como consecuencia de realizar el escaneo sin tener una API key de la base de datos NVD. En conclusión, **Trivy se posiciono como la herramienta más eficiente.**

Tabla 7. Tiempos de ejecución de las herramientas.

	Semgrep	**Scan**	**Dependency**	**Trivy**	**OSV**
Benchmark	375 s	299 s	-	-	-
PyGoat	-	-	2297 s	21 s	41 s

Luego de realizadas todas las evaluaciones Semgrep fue la que obtuvo mayor afinidad y rendimiento para SAST y Trivy para SCA. En este sentido, fueron las seleccionadas para integrar en los proyectos en Azure DevOps del Grupo Éxito.

4.4. Integración de las herramientas seleccionadas

El Grupo Éxito gestiona el SDLC a través de computación en la nube con Azure DevOps como SaaS. En esta plataforma la organización tiene varios proyectos entre los que se destacan 2; GCIT (Gerencia Corporativa Informática y Tecnología) con documentación de arquitectura y plantillas para pipelines de CI/CD. Y GCIT-Agile en donde se gestiona el portafolio de aplicaciones con base a la cultura DevOps.

El acceso a los repositorios para agregar y actualizar información en esta plataforma se gestiona mediante la utilidad llamada equipos, en donde se agregan personas con determinados roles que operaran bajo los mismos permisos. Por consiguiente, para la realización del piloto de pruebas, desde el área de arquitectura se concedieron los permisos necesarios para realizar cambios en los repositorios arquitectura-documentacion del proyecto GCIT y master-data-api-oke del

proyecto GCIT-Agile. En arquitectura-documentacion se creó una feature llamada integración-escaneres-seguridad a partir de la rama master. Esta misma feature se creó a partir de la rama develop en master-data-api-oke. Lo anterior porque desde la rama feature del repositorio piloto se consume la plantilla para integración y despliegue continuo del pipeline alojada en arquitectura-documentacion. El proceso descrito se realizó considerando la metodología Gitflow que fomenta la creación de features, para agregar nuevas funcionalidades al desarrollo, sin afectar el código que está en develop o master.

La Figura 20 presenta una visión general de cómo se consume el archivo que contiene las herramientas configuradas para la ejecución de pruebas de seguridad en el pipeline de CI de master-data-api-oke, a partir de la feature del repositorio de arquitectura-documentación.

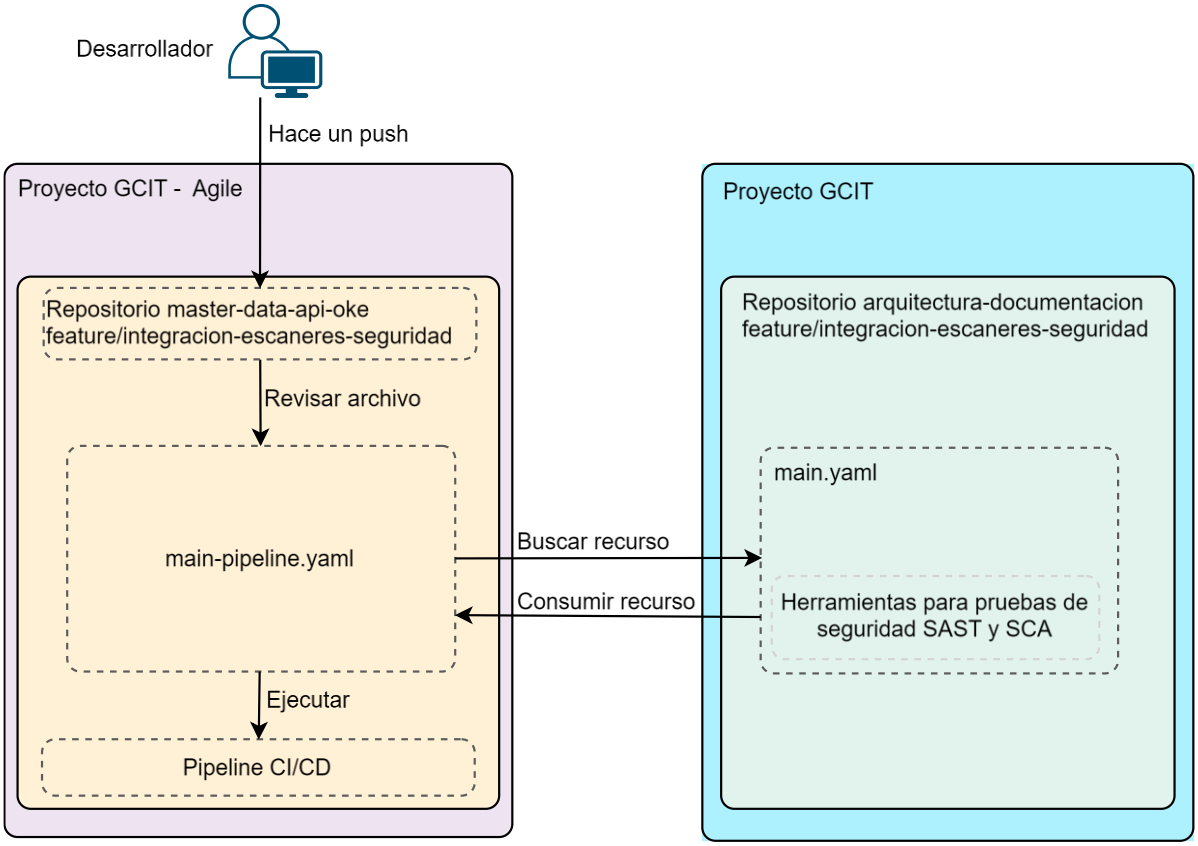


Figura 20. Flujo simplificado para la ejecución de pruebas de seguridad.

Inicialmente en la feature creada en master-data-api-oke se limpió el archivo main-pipeline.yaml. Luego se agregó la feature de arquitectura-documentacion como un recurso. En la figura 21 se muestra las configuraciones hechas para el recurso, variables y herencia.

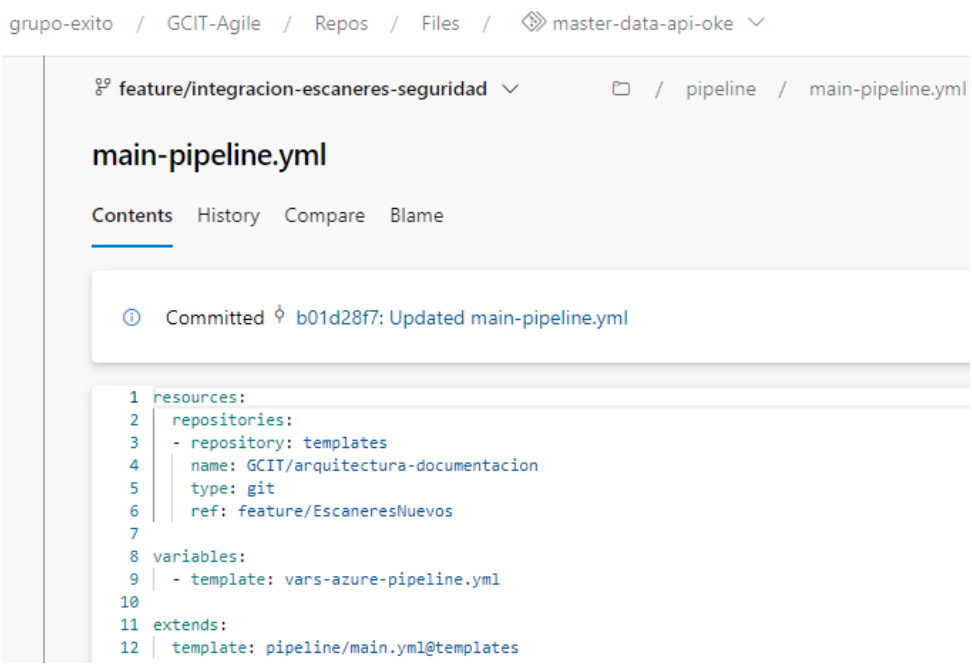


Figura 21. Configuración de recursos en la feature de GCIT-Agile.

Por otra parte, la figura 22 muestra la configuración del recurso interno en el archivo main.yaml del repositorio arquitectura-documentación, que permite acceder a la estructura del repositorio. El nombre y la rama de referencia se definen dinámicamente al consumir el archivo, utilizando variables de entorno para favorecer la flexibilidad y reutilización.

Figura 22. Configuración de recursos en la feature de GCIT.

Una vez definida la conexión entre el proyecto y la plantilla, el siguiente paso fue usar el código en la feature de master-data-api-oke de las aplicaciones Semgrep y Trivy, configuradas en secciones anteriores. Para conocer el código completo, diríjase al **Anexo D.**

4.5. Puntajes para ruptura del pipeline

En la tabla 8 se presentan las condiciones para romper el pipeline a partir de los resultados obtenidos por Semgrep. No solo se considera la severidad, sino que también el impacto y la probabilidad de explotación. Semgrep maneja 3 niveles de severidad error, warning e info, pero para priorizar se considera solo vulnerabilidades con severidad error y warning. A pesar de que las vulnerabilidades de menor riesgo no rompen el pipeline, si son reportadas en los informes. Sin embargo, su tratamiento pasa a una segunda instancia, esto implica que se asume cierto nivel de riesgo, para evitar consumir recursos en casos que pueden estar más dirigidos a buenas prácticas, falsos positivos y que no son un riesgo de alto impacto.

Tabla 8. Casos de rompimiento del pipeline con hallazgos de Semgrep.

Severidad	Impacto	Probabilidad
Error (High)	Todos los casos	Todos los casos
Warning (Medium)	High	Medium
	High	High

Trivy agrupa las vulnerabilidades según su severidad en alguna de las siguientes categorías: critical, high, medium, low y unknown. Sin embargo, para la ruptura del pipeline solo se tendrán en cuenta las vulnerabilidades con riesgo critical y high con los puntajes descritos en la tabla 9 del estándar CVSS. Esta medida se adopta en función de priorizar las exposiciones con mayor impacto para optimizar recursos.

Tabla 9. Casos de rompimiento del pipeline con hallazgos Trivy.

Severidad	CVSS v3.1
Critical	9.0 - 10.0
High	7.0 - 8.9

5. Resultados y análisis

5.1. Resultados de las pruebas con las herramientas integradas

Finalmente, se examinó master-data-api-oke. Los resultados indicaron que esta aplicación no tenía problemas relacionados con dependencias. Sin embargo, se detectaron 29 alarmas debidas a configuraciones incorrectas de IAC. Por otra parte, el pipeline se rompió gracias a los umbrales de ruptura establecidos en los ajustes de las herramientas, frente a casos con severidad crítica y alta.

En la figura 23 se observa la estructura del reporte de hallazgos generado por la extensión de Trivy. En la vista general se enumeran el total de problemas, vulnerabilidades en las librerías y secretos. Adicionalmente para cada alerta se muestra información relevante como el grado de severidad, ID, descripción del inconveniente y la localización de archivo afectado.

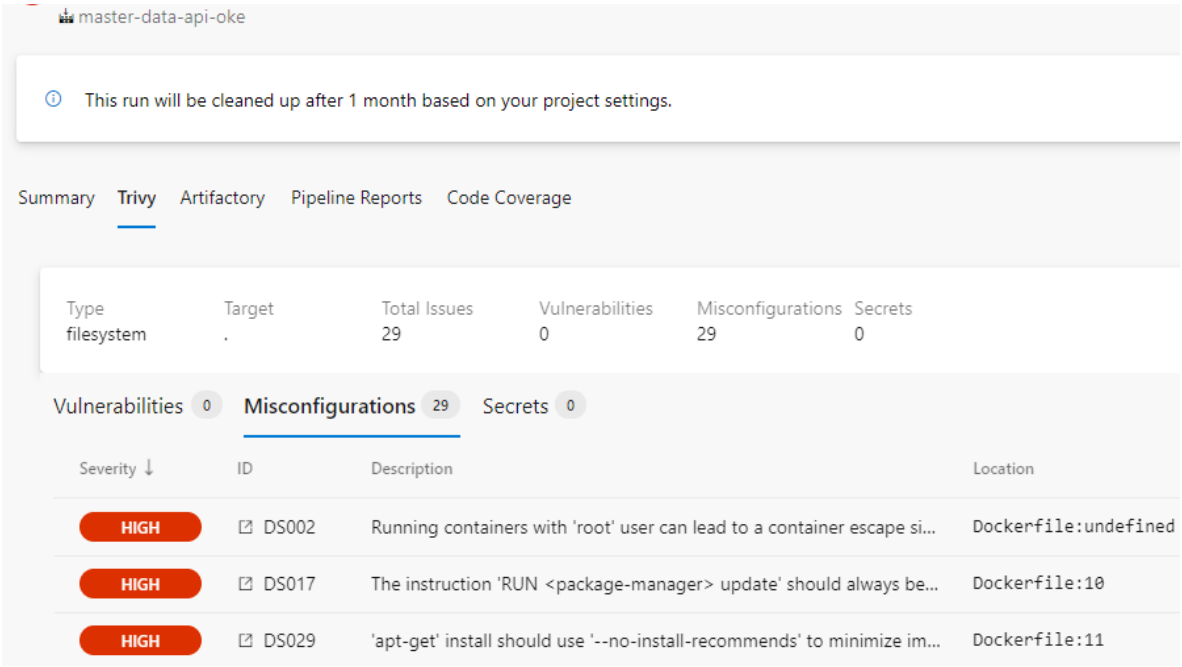


Figura 23. Reporte de los Hallazgos de Trivy.

En cuanto a los resultados para SAST Semgrep reporto 2 hallazgos que se muestran en la figura 24. El informe se exhibe información de utilidad para la investigación y posible mitigación de la vulnerabilidad incluido un hipervínculo hacia la regla de Semgrep (donde hay mucha más información de referencia), el CWE asociado y el archivo afectado.

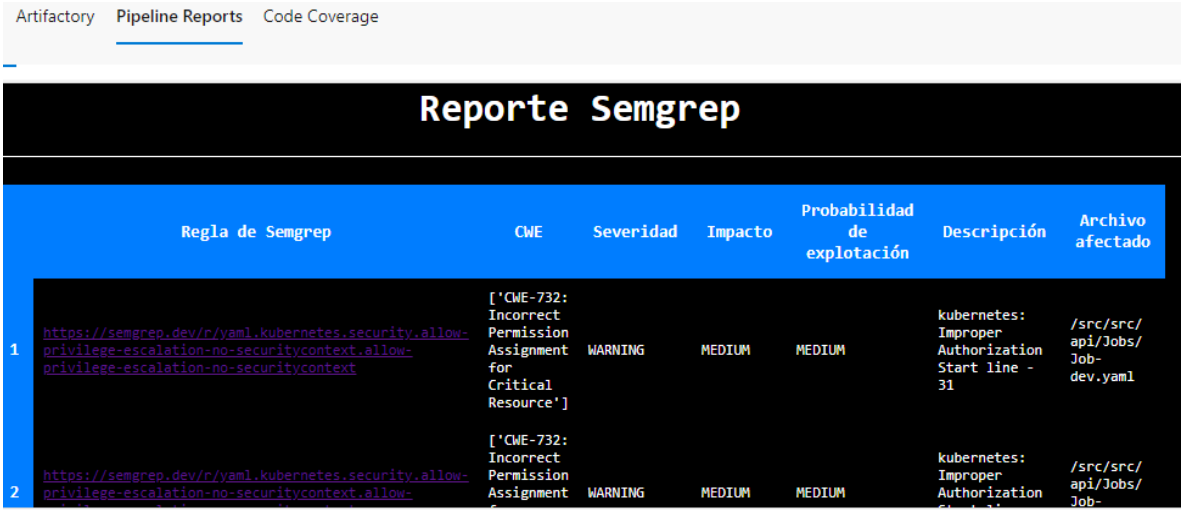


Figura 24. Reporte de los hallazgos de Semgrep.

5.2. Gestión de vulnerabilidades sin parche.

Cuando se encuentra una vulnerabilidad con Trivy, puede suceder que al momento de gestionarla no existe parche o una solución que pueda aplicarse en el tiempo previsto para su remediación. En este contexto, surge la necesidad de agregar la vulnerabilidad a una lista de aceptación, pero antes de agregarla, se deben estudiar dos factores relevantes: el impacto y la probabilidad de explotación para evaluar la viabilidad de esta acción.

En Trivy, el parámetro --ignorefile permite utilizar un archivo llamado trivyignore para excluir vulnerabilidades del reporte, lo que resulta útil, ya que permite configurar una fecha de expiración. Una vez que se cumple esta fecha, la vulnerabilidad sigue el proceso descrito en la figura 25. La ventaja de realizar esta operación es que evita romper los pipelines innecesariamente ante un incidente sin remediación conocida en el momento. Sin embargo, la desventaja es que las aplicaciones desarrolladas llevarán esa vulnerabilidad a producción hasta que se encuentren nuevos parches en alguna actualización de la lista de aceptación. Este es un riesgo informado que se debe se asume en función de las necesidades del negocio.

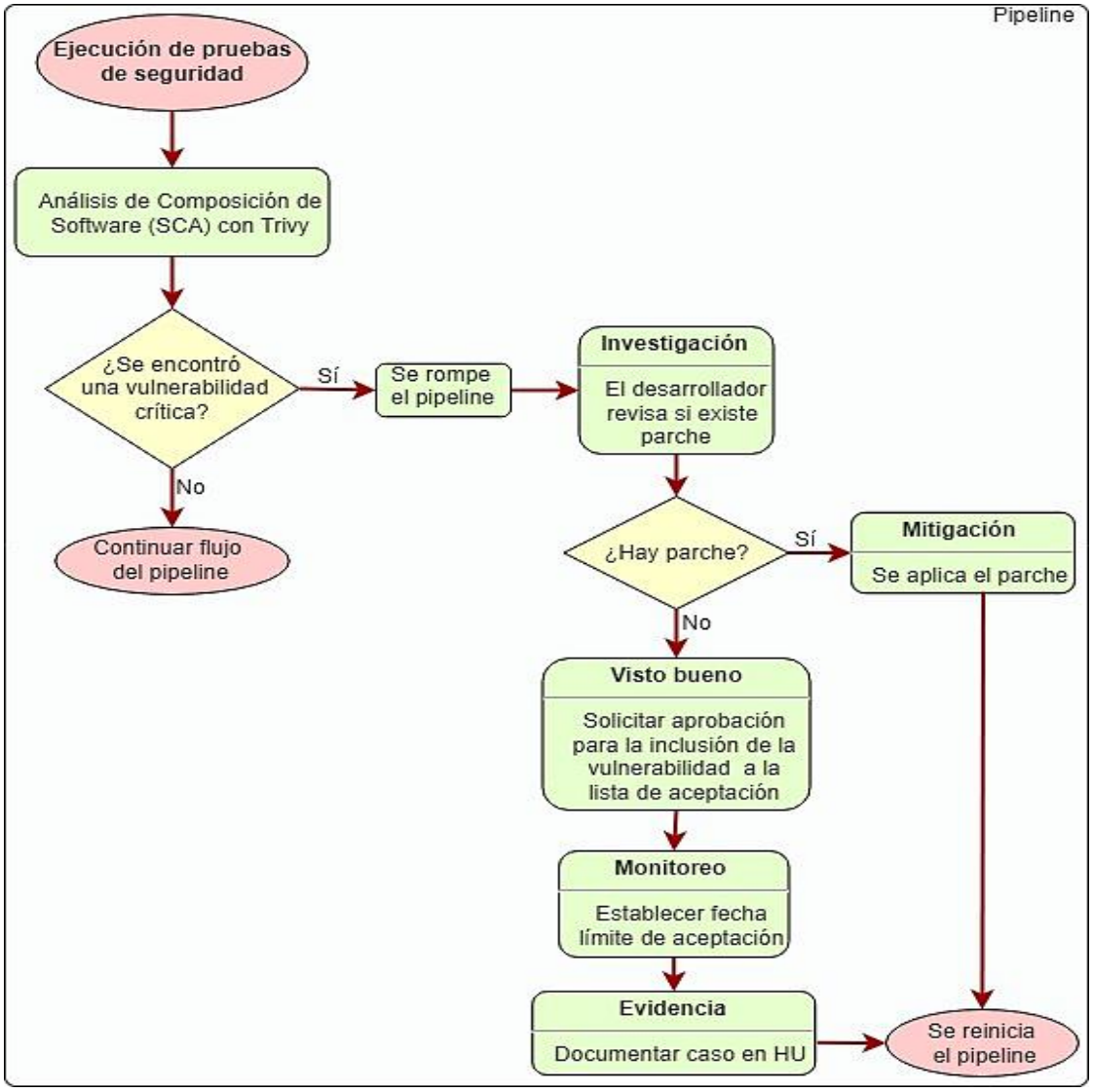


Figura 25. Flujo simplificado para el tratamiento de vulnerabilidades sin parche.

6. Conclusiones

El propósito fundamental de este estudio fue evaluar herramientas open source que pudieran integrarse en Azure DevOps, para fortalecer las prácticas de seguridad a través del análisis del código fuente. En este contexto, se realizó una investigación exhaustiva de herramientas, se evaluaron requisitos de aceptación y realizaron pruebas de rendimiento. De este modo se seleccionaron las herramientas que mejor se ajustaran a los requisitos del proyecto.

Se investigó en foros de ciberseguridad, conferencias y repositorios populares de GitHub para conocer más a profundidad la oferta de herramientas disponibles en el mercado. En esta búsqueda se encontraron 5 aplicaciones atractivas para el proyecto, que luego se evaluaron con criterios de aceptación y pruebas de rendimiento. En los criterios de aceptación se evaluaron factores como facilidad de integración con pipelines de CI en Azure DevOps, soporte de los principales lenguajes de desarrollo en el Grupo Éxito, personalización de reglas y actualizaciones frecuentes. Por otra parte, fue necesario crear una cuenta de Azure DevOps con una suscripción gratuita para hacer pruebas de precisión, exactitud y sensibilidad sobre 2 aplicaciones web vulnerables. A partir de los resultados de esas pruebas se identificó a Trivy como la mejor opción para SCA y a Semgrep para SAST. Luego, estas dos herramientas se integraron a la plantilla del Grupo Éxito, configuradas para interrumpir el pipeline por grado de severidad según unas tablas de priorización, generar reportes y publicarlos para su visualización de manera sencilla. También, se discutió acerca del tratamiento de vulnerabilidades sin parche y se describió el proceso a seguir ante este tipo de casos.

Generalmente se busca una herramienta que realice varios tipos de pruebas de seguridad para reducir el nivel de complejidad de las configuraciones en los entornos de desarrollo, sin embargo, como se vio en los resultados de las pruebas de rendimiento esa no siempre es la mejor opción. En el caso especifico de Scan los resultados no fueron los esperados a pesar de que usa otros escáneres para realizar el análisis del código. En otras palabras, este tipo de integraciones hacen más compleja y poco precisa la herramienta porque no se pueden configurar parámetros de escaneo según las necesidades del usuario, en su lugar, ofrece una configuración global poco especializada.

La precisión de las herramientas que soportan múltiples lenguajes de programación es limitada, es decir, los hallazgos suelen ser más acertados en determinados lenguajes. Por eso, la prueba de concepto se realizó sobre aplicaciones Java, Python y C# para priorizar los lenguajes usados en la dirección de TI. Otro aspecto importante a tener en cuenta es revisar cuidadosamente la documentación para conocer las capacidades de las herramientas, por ejemplo, OSV es un escáner relativamente nuevo con mucho potencial, pero con características experimentales que causan un poco de desconfianza en términos de estabilidad. Otro ejemplo es Dependency Check que a pesar de tener muchos años en el mercado, sus métodos de identificación de amenazas no se acoplan de la mejor manera a las necesidades actuales de la compañía. Debido a la necesidad de usar la API de la base de datos para que los escaneos sean rápidos.

Finalmente, la implementación de herramientas open source debe hacerse de manera razonable. No se trata solo de usarlas, sino de conocerlas a profundidad para obtener el máximo rendimiento y proteger la información confidencial. Una mala gestión puede poner en riesgo los sistemas de una empresa, por lo que se recomienda verificar que la herramienta realice los análisis localmente sin necesidad de enviar código a servidores remotos, que se le puedan configurar parámetros para desactivar él envió de métricas y que las fuentes que usa sean oficiales. El riesgo siempre existirá, no hay herramientas open source perfectas que detecten todas las vulnerabilidades y no pueden ser comparadas con herramientas comerciales porque el soporte, capacidades y mejoras están sujetas a la disponibilidad de los desarrolladores o de los aportes que la comunidad quiera hacer, en consecuencia, es necesario el monitoreo constante para verificar que estén funcionando de la forma esperada en la detección de riesgos. Además, es importante tener un plan de acción ante la posible comercialización de la herramienta open source por parte de sus creadores.

7. Póster

Para la presentación de este trabajo se realizó el póster disponible en siguiente página.

Departamento de Ingeniería Electrónica y de Telecomunicaciones

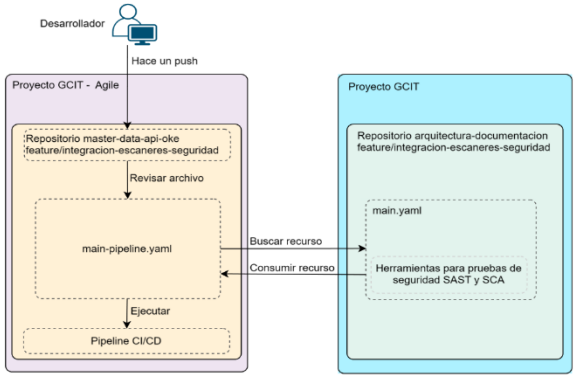
Investigación y evaluación de herramientas Open Source para escaneo de vulnerabilidades en el ciclo de vida de las aplicaciones, con el fin de ser integradas en los Pipelines en Azure del Grupo Éxito

UNIVERSIDAD DE ANTIOQUIA
1803
Facultad de Ingeniería

PRACTICANTE: David Felipe Vélez Cadavid
ASESORES: Sebastián Isaza Ramírez
Juan Fernando Hincapié Zapata

PROGRAMA: Ingeniería Electrónica
Semestre de la práctica: 2024-1

En este proyecto se investigaron e implementaron herramientas open source para realizar pruebas de seguridad del código fuente en los pipelines de CI en Azure. Estas herramientas generaron un impacto positivo en la compañía en relación automatización de pruebas, autonomía en el control de las herramientas y ahorro de recursos al prescindir de una plataforma comercial.



INTRODUCCIÓN

Los ciberdelincuentes se sienten atraídos por las compañías de la industria retail debido a la posibilidad latente de vulnerar sus sistemas a través de las transacciones que realizan los colaboradores, clientes y proveedores. De allí la necesidad de fortalecer las pruebas de seguridad con herramientas open source en el SDLC que permitan proteger la información de los clientes, los activos y los sistemas de la compañía.

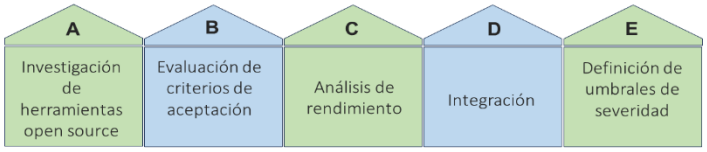
OBJETIVOS

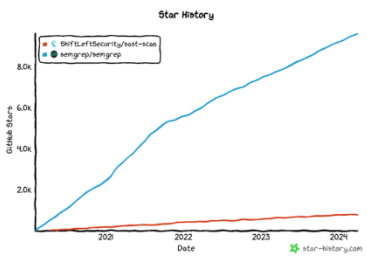
Fortalecer el sistema de pruebas de seguridad en el ciclo DevSecOps, mediante la investigación y la implementación de un piloto de pruebas para la evaluación de herramientas de aseguramiento open source, que puedan ser integradas en el ciclo de entregas continuas de software.

Identificar herramientas open source que sean adecuadas para la verificación continua de código, con el propósito de asegurar los repositorios alojados en Azure DevOps.	Definir los requisitos y criterios de aceptación enfocados en la escalabilidad, rendimiento y reducción de falsos positivos, frente a los cuales se depuren y seleccionen las herramientas open source investigadas.	Implementar un piloto de pruebas con las herramientas identificadas, integrándolas en la plantilla de un pipeline, para la evaluación de su funcionamiento en relación con el escaneo y alertamiento de vulnerabilidades en un repositorio.

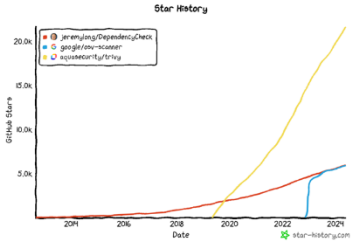
METODOLOGÍA

La metodología utilizada se compone de 5 etapas:





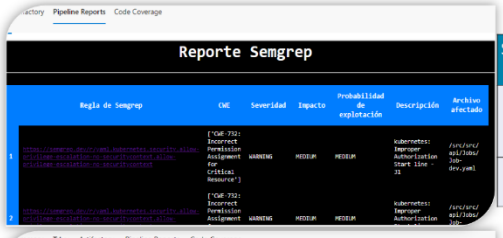
SAST (Static Application Security Testing)



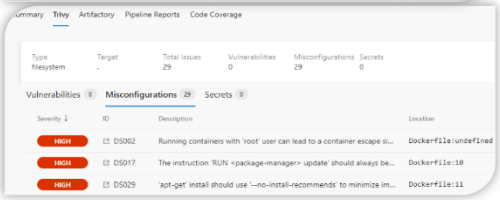
SCA (Software Composition Analysis)

RESULTADOS

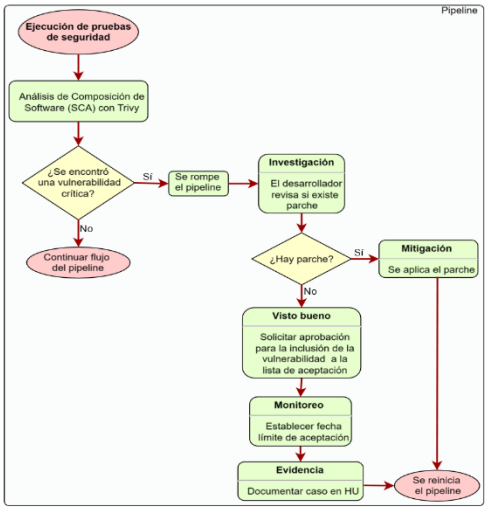
Las herramientas que se integraron a la plantilla de arquitectura de la dirección de TI del Grupo Éxito fueron **Semgrep** para SAST, con una capacidad de descubrir código vulnerable en Java del 97.81% y **Trivy** para SCA con capacidad del 100% en código Python. Estas herramientas permitieron el análisis, generación de reportes y la interrupción del pipeline ante vulnerabilidades críticas que se encontraron en un repositorio de pruebas de la compañía.



Severidad SAST	Impacto	Probabilidad
Error	Todos los casos	Todos los casos
Warning	High	Medium
	High	High



Severidad SCA	CVSS v3.1
Critical	9.0 - 10.0
High	7.0 - 8.9



Tratamiento de vulnerabilidades sin parche

CONCLUSIONES

Herramientas que soportan múltiples lenguajes de programación no son igual de precisas en todos ellos. Así mismo, las herramientas que realizan varias pruebas de seguridad como Scan no tienen muchas opciones de configuración, en su lugar, cuentan con una configuración global poco especializada.

No siempre las herramientas nuevas o las más antiguas son la mejor opción. Ejemplo de ello es OSV que es reciente, pero con características experimentales y Dependency Check que es antigua, pero con métodos poco sofisticados y lentos para realizar los escaneos.

Las herramientas open source no pueden ser comparadas con herramientas comerciales, debido a que su nivel de soporte, capacidades y mejoras están sujetas a la disponibilidad de la comunidad. Por esta razón, es importante monitorearlas y actualizarlas contantemente para reducir el riesgo de fallas en las mismas.

DATOS DE CONTACTO DEL AUTOR:

8. Referencias Bibliográficas

[1] Atlassian, "DevOps," Atlassian. [En línea]. Disponible: https://www.atlassian.com/es/devops.

[2] Oracle Colombia, "¿Qué es DevOps?," Oracle. [En línea]. Disponible: https://www.oracle.com/co/devops/what-is-devops/.

[3] J. A. Soto Velásquez, "ADAM: Método Ágil para Adopción de estrategias de DevOps," Tesis de Maestría, Escuela de Ciencias Aplicadas e Ingeniería, Universidad EAFIT, Medellín, Colombia, 2023. [En línea]. Disponible en: https://repository.eafit.edu.co/server/api/core/bitstreams/4e74a52f-bdaf-4e31-9ada-52ed37b80877/content

[4] F. Melo, «¿Git? ¿GiT Flow? ¿GitHub Flow? ¿Trunk? ¿Ship? - Fernando Melo - Medium», Medium, 25 de julio de 2023. [En línea]. Disponible en: https://fernandojmo.medium.com/git-git-flow-github-flow-trunk-ship-33e00b3d45fe.

[5] IBM, "¿Qué es DevOps?," IBM. [En línea]. Disponible: https://www.ibm.com/mx-es/topics/devops.

[6] L. S.-M. Joaquín, «Introducción a DevSecOps para la mejora de los procesos de desarrollo de software con herramientas Open Source», 18 de junio de 2023. https://openaccess.uoc.edu/handle/10609/148098.

[7] "¿Qué es DevSecOps? - Explicación de las operaciones de seguridad para desarrolladores - AWS," Amazon Web Services, Inc. https://aws.amazon.com/es/what-is/devsecops/.

[8] "¿Qué es el modelado de amenazas? | Cloudflare." https://www.cloudflare.com/es-es/learning/security/glossary/what-is-threat-modeling/

[9] "¿Qué es DevSecOps? Definición y procedimientos recomendados | Seguridad de Microsoft." https://www.microsoft.com/es-mx/security/business/security-101/what-is-devsecops.

[10] Atlassian, "Herramientas de DevSecOps | Atlassian," Atlassian. https://www.atlassian.com/es/devops/devops-tools/devsecops-tools

[11] ShiftLeftSecurity, "ShiftLeftSecurity/sast-scan," GitHub. https://github.com/ShiftLeftSecurity/sast-scan.

[12] Semgrep, "Semgrep OSS philosophy," Semgrep Documentation. https://semgrep.dev/docs/contributing/semgrep-philosophy.

[13] Semgrep, "Supported languages," Semgrep Documentation. https://semgrep.dev/docs/supported-languages.

[14] Semgrep, "Explore," Semgrep. https://semgrep.dev/explore.

[15] J. Long, "File Type Analyzers," OWASP Dependency-Check. https://jeremylong.github.io/DependencyCheck/analyzers/index.html.

[16] J. Long, "Dependency-Check Internals," GitHub Pages. Available: https://jeremylong.github.io/DependencyCheck/general/internals.html.

[17] J. Long, "Dependency-Check Data," GitHub Pages. Available: https://jeremylong.github.io/DependencyCheck/data/index.html.

[18] OpenSSF, "OSV Schema," GitHub Pages. Available: https://ossf.github.io/osv-schema/.

[19] Google, "Current Data Sources," OSV.dev. Available: https://google.github.io/osv.dev/data/#current-data-sources.

[20] Google, "OSV.dev," Available: https://google.github.io/osv.dev/.

[21] Google, "Announcing a Unified Vulnerability Schema for Open Source," Google Online Security Blog. Available: https://google.github.io/osv.dev/.

[22] Google, "OSV.dev FAQ," Available: https://google.github.io/osv.dev/faq/. [

[23] Microsoft, "Key Pipelines Concepts," Azure DevOps Documentation. Available: https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/key-pipelines-concepts?view=azure-devops.

[24] Sentrio, "¿Qué es Azure DevOps?," Sentrio Blog. Disponible en: https://sentrio.io/blog/que-es-azure-devops/.

[25] ,Red Hat, "¿Qué es YAML?," Available: https://www.redhat.com/es/topics/automation/what-is-yaml.

[26] Aqua Security, "Trivy Vulnerability Scanner Documentation," Available: https://aquasecurity.github.io/trivy/v0.52/docs/scanner/vulnerability/.

[27] FIRST, "CVSS v3.1 Specification Document," Available: https://www.first.org/cvss/v3.1/specification-document.

[28] Aqua Security, "Trivy Misconfiguration Scanner Documentation," Available: https://aquasecurity.github.io/trivy/v0.52/docs/scanner/misconfiguration/.

[29] Aqua Security, "Trivy Secret Scanner Documentation," Available: https://aquasecurity.github.io/trivy/v0.52/docs/scanner/secret/.

[30] Google, "Third-Party Licenses," Available: https://opensource.google/documentation/reference/thirdparty/licenses.

[31] Aqua Security, "Trivy License Scanner Documentation," Available: https://aquasecurity.github.io/trivy/v0.52/docs/scanner/license/.

[32] Aqua Security, "Supported Languages - Trivy Documentation," Available: https://aquasecurity.github.io/trivy/v0.52/docs/coverage/language/#supported-languages

[33] Google, "Third-Party Licenses," Available: https://opensource.google/documentation/reference/thirdparty/licenses.

[34] Aqua Security, "Filtering Configuration - Trivy Documentation," Available: https://aquasecurity.github.io/trivy/v0.52/docs/configuration/filtering/.

[35] "Star History for GitHub Repositories," Star-History.com. [Online]. Available: https://star-history.com/#ShiftLeftSecurity/sast-scan&semgrep/semgrep&Date.

[36] "Star History for GitHub Repositories," Star-History.com. [Online]. Available: https://star-history.com/#jeremylong/DependencyCheck&google/osv-scanner&aquasecurity/trivy&Date.

[37] "Google Benchmark," GitHub. [Online]. Available: https://github.com/google/benchmark.

[38] "PyGoat," GitHub. [Online]. Available: https://github.com/adeyosemanputra/pygoat.

[39] "OWASP Benchmark Project," OWASP. [Online]. Available: https://owasp.org/www-project-benchmark/.

[40] "SAP Azure Pipelines HTML Report," Visual Studio Marketplace. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=ShahzebKhan.sap-azure-pipelines-html-report.

[41] "Trivy Official," Visual Studio Marketplace. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=AquaSecurityOfficial.trivy-official.

[42] Trustwave, "2023 Retail Sector Threat Briefing and Mitigation Strategies," Trustwave, 2023. [En línea]. Disponible en: https://www.trustwave.com/en-us/resources/library/documents/2023-retail-sector-threat-briefing-and-mitigation-strategies/.

9. Anexos

Anexo A: Código para la ejecución de pruebas SAST con herramientas preseleccionadas

En esta sección se trabajó con la aplicación vulnerable BenchmarkJAVA de OWASP.

```
trigger:
- master

jobs:
# Scan de Shiftleft.
#*****************************************
- job: 'Scan_SAST'
  displayName: 'Scan(SAST)'
  pool:
    name: 'Azure Pipelines'
    VmImage: Ubuntu-latest
  steps:
    - checkout: self
      persistCredentials: true
      fetchDepth: 0

    # Se descarga la imagen de Scan
    - task: Bash@3
      displayName: 'Descargando Scan'
      inputs:
        targetType: 'inline'
        script: docker pull shiftleft/sast-scan:latest

    # Se compila el código
    - task: Maven@3
      inputs:
        mavenPomFile: 'pom.xml'
        mavenOptions: '-Xmx3072m'
        goals: 'clean compile'

    # Se realiza el escaneo
    - task: Bash@3
      displayName: "Ejecutando Scan"
      inputs:
        targetType: 'inline'
        script: |
          docker run --rm -e "WORKSPACE=${PWD}" \
          -v ~/.m2:/.m2 \
          -v $(Build.SourcesDirectory):/app:cached \
          -v "$(Build.ArtifactStagingDirectory):/reports:cached" \
          shiftleft/scan scan --src /app/target/classes/org/owasp/benchmark/testcode \
          --type java \
          --out_dir /reports
      continueOnError: false

    # Se publican archivos con los hallazgos
    - task: PublishBuildArtifacts@1
      displayName: 'Publicando reportes'
      inputs:
        ArtifactName: 'ReporteScan(SAST)'
        PathtoPublish: '$(Build.ArtifactStagingDirectory)'
      condition: always()

# Semgrep OSS
#************************************
- job: Semgrep_SAST
  displayName: 'Semgrep(SAST)'
  pool:
```

```
  name: 'Azure Pipelines'
  vmImage: 'ubuntu-latest'

steps:
- checkout: self
  fetchDepth: 1

# Se descarga la imagen de Semgrep
- task: Bash@3
  displayName: 'Descargando Semgrep'
  inputs:
    targetType: 'inline'
    bash: docker pull semgrep/semgrep:latest

# Se parametriza la herramienta y se ejecuta
- task: Bash@3
  displayName: "Ejecutando Semgrep"
  inputs:
    targetType: 'inline'
    script: |
      docker run \
      -v "$(Build.SourcesDirectory):/src:cached" \
      -v "$(Build.ArtifactStagingDirectory):/reports:cached" \
      semgrep/semgrep:latest \
      semgrep scan \
      --config "p/r2c-security-audit" \
      --metrics=off \
      --pro \
      -q \
      --exclude-rule java.lang.security.audit.crypto.desede-is-deprecated.desede-is-deprecated \
      --exclude-rule java.lang.security.audit.cookie-missing-httponly.cookie-missing-httponly \
      /src/src/main/java/org/owasp/benchmark/testcode \
      --json-output /reports/reporte-semgrep.json
      report_json="$(Build.ArtifactStagingDirectory)/reporte-semgrep.json"

      # Verificación del formato del reporte
      if [ ! jq . "$report_json" > /dev/null 2>&1 ]; then
        echo "NO se encontro reporte en formato JSON o el reporte no se formateó correctamente."
        exit 1
      fi

      # Manejo de errores para jq
      jq_validation() {
          jq "$@" "$report_json" || {
              echo "Escaneo de JSON Fallo. Revisar jerarquía y nombre de variables."
              exit 1
          }
      }

      severity_warning=$(jq_validation '[.results[]?
          | select(.severity == "WARNING" and (.extra.impact == "HIGH" and
          (.extra.likelihood == "MEDIUM" or .extra.likelihood == "HIGH" )))] | length')
      results_len=$(jq_validation '.results | length')

      # Ruptura del pipeline ante Severidad ERROR
      if grep -q '"severity": "ERROR"' "$report_json"; then
        echo "Se hallaron vulnerabilidades con severidad ERROR."
        echo "Por favor revise el reporte de los hallazgos que le ayudará a identificarlas y solucionarlas."
        exit 1
      elif [ "$results_len" -eq 0 ]; then
          echo "No se encontraron vulnerabilidades."
      else
        echo "No se hallaron vulnerabilidades con Severidad ERROR"
      fi
      # Ruptura del pipeline ante severidad WARNING con Impacto High
```

```
    if [ "$severity_warning" -gt 0 ]; then
      echo "Se encontraron vulnerabilidades con severidad WARNING e impacto alto."
      echo "Por favor revise el reporte de los hallazgos que le ayudará a identificarlas y solucionarlas."
      exit 1
    elif [ "$results_len" -eq 0 ]; then
       echo "No se encontraron vulnerabilidades."
    else
      echo "No se encontraron vulnerabilidades con severidad WARNING e impacto alto"
    fi
  continueOnError: "false"

 # Se instalan las librerías necesarias para generar un archivo html.
- task: Bash@3
  displayName: 'Instalación de dependencias'
  inputs:
    targetType: 'inline'
    script: pip install --upgrade pandas jinja2
  condition: always()

# Se construye un archivo html a partir del json.
- task: PythonScript@0
  displayName: 'Generando reporte'
  inputs:
    scriptSource: 'inline'
    script: |
      import json
      import pandas as pd
      from urllib.parse import quote
      try:
          with open('$(Build.ArtifactStagingDirectory)/reporte-semgrep.json', 'r') as d:
            findings = json.load(d)
          if 'results' in findings and len(findings['results']) >0:
            data = []
            for finding in findings['results']:
                check_id = finding.get('check_id', '')

                # Se muestra la ruta del archivo en varias lineas
                path = finding.get('path', '')
                long = len(path)
                ind_div1 = path.find('/', long // 4)
                ind_div2 = path.find('/', 2 * long // 4)
                ind_div3 = path.find('/', 3 * long // 4)
                if ind_div1 != -1 and ind_div2 != -1 and ind_div3 != -1:
                    path = path[:ind_div1+1] + '<br>' + path[ind_div1+1:ind_div2+1] + '<br>' + path[ind_div2+1:ind_div3+1] + '
                            <br>' + path[ind_div3+1:]
                elif ind_div1 != -1 and ind_div2 != -1:
                    path = path[:ind_div1+1] + '<br>' + path[ind_div1+1:ind_div2+1] + '<br>' + path[ind_div2+1:]
                elif ind_div1 != -1:
                    path = path[:ind_div1+1] + '<br>' + path[ind_div1+1:]
                else:
                    path = path
                # Se define el contenido de las variables que se mostrarán en la tabla del HTML
                start_line = finding.get('start', {}).get('line', '')
                description = finding.get('extra', {}).get('message', '')
                impact = finding.get('extra', {}).get('metadata', {}).get('impact', '')
                category = finding.get('extra', {}).get('metadata', {}).get('category', '')
                cwe_value = finding.get('extra', {}).get('metadata', {}).get('cwe', '')
                if isinstance(cwe_value, list):
                    cwe = ', '.join(cwe_value)
                else:
                    cwe = cwe_value
                likelihood = finding.get('extra', {}).get('metadata', {}).get('likelihood', '')
                vulnerability_class = ', '.join(finding.get('extra', {}).get('metadata', {}).get('vulnerability_class', ''))
                technology = ', '.join(finding.get('extra', {}).get('metadata', {}).get('technology', ''))
                severity = finding.get('extra', {}).get('severity', '')
```

```
    references = ', '.join(finding.get('extra', {}).get('metadata', {}).get('references', ''))

    # Se muestra la url de la regla de Semgrep en varias lineas
    url = "https://semgrep.dev/r/" + quote(check_id)
    long_url = len(url)
    url_div1 = url.find('/', long_url // 4)
    url_div2 = url.find('/', 2 * long_url // 4)
    url_div3 = url.find('/', 3 * long_url // 4)
    if url_div1 != -1 and url_div2 != -1 and url_div3 != -1:
        url_mult = url[:url_div1+1] + '<br>' + url[url_div1+1:url_div2+1] + '<br>' + url[url_div2+1:url_div3+1] +
                   '<br>' + url[url_div3+1:]
    elif url_div1 != -1 and url_div2 != -1:
        url_mult = url[:url_div1+1] + '<br>' + url[url_div1+1:url_div2+1] + '<br>' + url[url_div2+1:]
    elif url_div1 != -1:
        url_mult = url[:url_div1+1] + '<br>' + url[url_div1+1:]
    else:
        url_mult = url

    # Se especifican las variables y la información que se mostrarán en el HTML
    data.append({
        'Regla de Semgrep': f"<a href={url} target='_blank'>{url_mult}</a>",
        'CWE': cwe_value,
        'Severidad': severity,
        'Impacto': impact,
        'Probabilidad de explotación': likelihood,
        'Descripción': f"{technology}: {vulnerability_class} Start line - {start_line}",
        'Archivo afectado': path})
df = pd.DataFrame(data)
# Se define el estilo que tendrá la tabla en el archivo HTML
styles = [
    dict(selector="tr:hover", props=[("background-color", "#121614")]),
    dict(selector="table", props=[("margin", "auto"), ("width", "100%"), ("font-family", "Arial, sans-serif"),
    ("background-color", "#334139")]),
    dict(selector="th", props=[("font-size", "14px"), ("background-color", "#007bff"), ("color", "white"), ("padding",
    "12px"), ("text-align", "center")]),
    dict(selector="td", props=[("overflow-wrap", "break-word"), ("font-size", "12px"), ("text-align", "left"),
    ("padding", "8px")]),
]
df.index = df.index + 1
html_table = (df.style
        .set_table_styles(styles).to_html(index=False, escape=False))
html_page = f"""
<!DOCTYPE html>
<html>
<head>
    <title>Reporte Semgrep</title>
    <style>
        body {{
            background-color: #000000;
            color: #ffffff;
            font-family: "Overpass Mono", monospace;
        }}
        h1 {{
            text-align: center;
            color: #ffffff;
            padding: 20px;
            font-size: 2em;
            border-bottom: 1px solid #ffffff;
        }}
        table {{
            margin: auto;
            border-collapse: collapse;
            width: 80%;
            font-size: 16px;
            color: #ffffff;
```

```
                }}
                th, td {{
                    padding: 8px;
                }}
                tr:hover, td:hover {{
                    background-color: #090B0A;
                }}
                th {{
                    padding-top: 12px;
                    padding-bottom: 12px;
                    text-align: left;
                    background-color: #4CAF50;
                    color: white;
                }}
            </style>
        </head>
        <body>
            <h1>Reporte Semgrep</h1>
            {html_table}
        </body>
        </html>
        """
        # Se guarda el archivo HTML con estilo CSS aplicado
        with open('$(Build.ArtifactStagingDirectory)/reporte-semgrep.html', 'w') as d:
            d.write(html_page)
      else:
        message= f"<html><head><body></html><h2>No se encontraron vulnerabilidades de
                  SAST.</h2></head></body>"
        with open('$(Build.ArtifactStagingDirectory)/reporte-semgrep.html', 'w') as f:
          f.write(message)
        print('No se encontraron vulnerabilidades de SAST.')
    except Exception as e:
      message= f"<html><head></head><body><h2>No se encontraron vulnerabilidades de SAST.</h2></body></html>"
      with open('$(Build.ArtifactStagingDirectory)/reporte-semgrep.html', 'w') as f:
        f.write(message)
      print(f"Error: {e}")
  environmentVariable: 'PYTHONPATH'
 condition: always()

# Se publican los resultados de la prueba
- task: PublishHtmlReport@3
 displayName: 'Publicando reporte'
 inputs:
  tabName: 'Reporte Semgrep'
  reportDir: '$(Build.ArtifactStagingDirectory)/reporte-semgrep.html'
 condition: always()

# Se publica el archivo html con los hallazgos
- task: PublishBuildArtifacts@1
 displayName: 'Publicando reportes'
 inputs:
  ArtifactName: 'ReporteSemgrep(SAST)'
  PathtoPublish: '$(Build.ArtifactStagingDirectory)/reporte-semgrep.html'
 condition: always()

# Limpieza de los archivos temporales de ArtifactStagingDirectory
- task: DeleteFiles@1
 displayName: 'Limpiando ArtifactStagingDirectory'
 inputs:
  SourceFolder: '$(Build.ArtifactStagingDirectory)'
  Contents: |
   reporte-semgrep.json
  removeSourceFolder: false
  cleanTargetFolder: false
 condition: always()
```

Anexo B: Código para la ejecución de pruebas SCA con herramientas preseleccionadas

En esta sección se trabajó con la aplicación vulnerable PyGoat.

```
trigger:
- master

jobs:
# Dependency Check
#**************************************
- job: 'DependencyCheck_SCA'
  displayName: 'DependencyCheck(SCA)'
  pool:
    name: 'Azure Pipelines'
    vmImage: 'ubuntu-latest'

  steps:
    - checkout: self
      fetchDepth: 1

    - task: Bash@3
      displayName: 'Ejecutando DependencyCheck'
      inputs:
        targetType: 'inline'
        script: |
          mkdir -p $(Build.ArtifactStagingDirectory)/report
          chmod 777 $(Build.ArtifactStagingDirectory)/report
          docker run --rm \
          -v "$(Build.SourcesDirectory):/src:cached" \
          -v "$(Build.ArtifactStagingDirectory)/report:/reports:cached" \
          owasp/dependency-check:latest \
          --scan /src \
          --format "HTML" \
          --project "OWASP Dependency Check" \
          --enableExperimental \
          --out /reports \
          --failOnCVSS 7.0
      continueOnError: false
    - task: PublishHtmlReport@3
      displayName: 'Publicando reporte'
      inputs:
        tabName: 'Reporte Dependency Check'
        reportDir: '$(Build.ArtifactStagingDirectory)/report/dependency-check-report.html'
      condition: always()

    # Publicación de artefacto html
    - task: PublishBuildArtifacts@1
      displayName: 'Publicando reportes'
      inputs:
        artifactName: 'Reporte-DependencyCheck(SCA)'
        pathtoPublish: '$(Build.ArtifactStagingDirectory)/report'
      condition: always()

# Trivy
# #*********************************
- job: 'Trivy_SCA'
  displayName: 'Trivy(SCA)'
  pool:
    name: 'Azure Pipelines'
    VmImage: Ubuntu-latest
  steps:
    - checkout: self
      fetchDepth: 1

    # Configuración de los parámetros de escaneo Trivy
```

```
    - task: trivy@1
      displayName: 'Ejecutando Trivy'
      inputs:
        options: --scanners "vuln,misconfig,secret,license" --exit-code 1
        version: latest
        ignoreUnfixed: false
        path: .
      continueOnError: false

# OSV - Scanner
# #**********************************
- job: OsvScanner_SCA
  displayName: 'OSV-Scanner(SCA)'
  pool:
    name: 'Azure Pipelines'
    vmImage: 'ubuntu-latest'

  steps:
  - checkout: self
    fetchDepth: 1

  # Instalación de OSV-Scanner
  - task: Bash@3
    displayName: 'Instaciación de OSV - Scanner'
    inputs:
      targetType: 'inline'
      script: docker pull ghcr.io/google/osv-scanner:latest

  # Ejecución de OSV-Scanner
  - task: Bash@3
    displayName: 'Ejecutando OSV - Scanner'
    inputs:
      targetType: 'inline'
      script: |
        docker run \
        -v "$(Build.SourcesDirectory):/src:cached" \
        -v "$(Build.ArtifactStagingDirectory):/reports:cached" ghcr.io/google/osv-scanner \
        --verbosity verbose \
        --format json --output /reports/reporte-osv-scanner.json \
        -r /src
        report_json="$(Build.ArtifactStagingDirectory)/reporte-osv-scanner.json"

        # Verificación del formato del reporte
        if [ ! jq . "$report_json" > /dev/null 2>&1 ]; then
          echo "NO se encontro reporte en formato JSON o el reporte no se formateó correctamente."
          exit 1
        fi
        # Manejo de errores para jq
        jq_validation() {
            jq "$@" "$report_json" || {
                echo "Escaneo de JSON Fallo. Revisar jerarquía y nombre de variables"
                exit 1
            }
        }
        max_severity=$(jq_validation '[.results[]?
          | .packages[]?
          | .groups[]?
          | select(.max_severity != null and (.max_severity | try tonumber // empty) > 7.9)] | length')

        results_len=$(jq_validation '.results | length')

        # Ruptura del pipeline ante hallazgos con severidad mayor a 7.9
        if [ "$max_severity" -gt 0 ]; then
            echo "Se encontraron vulnerabilidades con severidad mayor a 7.9."
            echo "Por favor revise el reporte HTML que le ayudará a identificarlas y solucionarlas."
```

```
        exit 1
    elif [ "$results_len" -eq 0 ]; then
            echo "No se encontraron vulnerabilidades."
    else
        echo "No se encontraron vulnerabilidades con severidad mayor a 7.9"
    fi
  continueOnError: "false"
# Publicar los resultados como un artefacto
- task: PublishBuildArtifacts@1
  displayName: 'Publicando reporte .json'
  inputs:
    PathtoPublish: '$(Build.ArtifactStagingDirectory)/reporte-osv-scanner.json'
    ArtifactName: 'Reporte-OSV-Scanner(SCA)'
  condition: always()

- task: Bash@3
  displayName: 'Instalación de dependencias'
  inputs:
    targetType: 'inline'
    script: pip install --upgrade jinja2 pandas
  condition: always()

# Generación de reporte en HTML
- task: PythonScript@0
  displayName: 'Generando reporte HTML'
  inputs:
    scriptSource: 'inline'
    script: |
      import json
      import pandas as pd
      from urllib.parse import quote
      try:
          with open('$(Build.ArtifactStagingDirectory)/reporte-osv-scanner.json', encoding='latin-1') as d:
            findings = json.load(d)
          if 'results' in findings and len(findings['results']) >0:
            data = []
            for finding in findings['results']:
                for package in finding.get('packages', []):
                    vulnerabilities = package.get('vulnerabilities', [])
                    if vulnerabilities:
                        pack = package.get('package', {}).get('name', '')
                        version = package.get('package', {}).get('version', '')
                        ecosystem = package.get('package', {}).get('ecosystem', '')
                        source = finding.get('source', {}).get('path', '')
                        url = 'https://osv.dev/vulnerability/'
                        for vulnerability, group in zip(vulnerabilities, package.get('groups', [])):
                            cve_alias = next((cve for cve in vulnerability.get('aliases', []) if cve.startswith("CVE")), '')
                            id = vulnerability.get('id', '')
                            severity = vulnerability.get('database_specific', {}).get('severity', '')
                            fix_version = vulnerability.get('affected', [{}])[0].get('ranges', [{}])[0].get('events', [{}])[1].get('fixed', '')
                            cvss = group.get('max_severity', '')
                            data.append({
                                'URL': f'<a href="{url + quote(id)}" target="_blank">{url + quote(id)}</a>',
                                'CVE': cve_alias,
                                'Severity': severity,
                                'CVSS': cvss,
                                'Package': pack,
                                'Source': source ,
                                'Ecosystem': ecosystem,
                                'Version': version ,
                                'Fix Version': fix_version})
            df = pd.DataFrame(data)
            def cvss_color(val):
                color = []
                if isinstance(val, pd.Series):
```

```
        for v in val:
            if v != '' and v is not None:
                if float(v) > 9.0:
                    color.append('color:red')
                else:
                    color.append('color:white')
            else:
                color.append('color:white')
    return color
df['CVSS'] = df['CVSS'].apply(lambda x: round(float(x), 1) if x else None)
styles = [
    dict(selector="tr:hover", props=[("background-color", "#121614")]),
    dict(selector="table", props=[("margin", "auto"), ("width", "100%"), ("font-family", "Arial, sans-serif"),
    ("background-color", "#334139")]),
    dict(selector="th", props=[("font-size", "14px"), ("background-color", "#007bff"), ("color", "white"), ("padding",
    "12px"), ("text-align", "center")]),
    dict(selector="td", props=[("font-size", "12px"), ("text-align", "left"), ("padding", "8px")])]
df.index = df.index + 1
html_table = (df.style
        .format({'CVSS': '{:.1f}'}, na_rep='')
        .apply(cvss_color, subset=['CVSS'])
        .set_table_styles(styles).to_html(index=False, escape=False))
html_page = f"""
<!DOCTYPE html>
<html>
<head>
    <title>Reporte OSV Scanner</title>
    <style>
        body {{
            background-color: #000000;
            color: #ffffff;
            font-family: "Overpass Mono", monospace;
        }}
        h1 {{
            text-align: center;
            color: #ffffff;
            padding: 20px;
            font-size: 2em;
            border-bottom: 1px solid #ffffff;
        }}
        table {{
            margin: auto;
            border-collapse: collapse;
            width: 80%;
            font-size: 16px;
            color: #ffffff;
        }}
        th, td {{
            padding: 8px;
        }}
        tr:hover, td:hover {{
            background-color: #090B0A;
        }}
        th {{
            padding-top: 12px;
            padding-bottom: 12px;
            text-align: left;
            background-color: #4CAF50;
            color: white;
        }}
    </style>
</head>
<body>
    <h1>Reporte OSV Scanner</h1>
    {html_table}
```

```
            </body>
            </html>
            """
            # Se guarda el archivo HTML con estilo CSS aplicado
            with open('$(Build.ArtifactStagingDirectory)/reporte-osv-scanner.html', 'w') as d:
                d.write(html_page)
          else:
            message= f"<html><head></head><body><h2>No se encontraron vulnerabilidades de SCA.</h2></body></html>"
            with open('$(Build.ArtifactStagingDirectory)/reporte-osv-scanner.html', 'w') as f:
              f.write(message)
            print('No se encontraron vulnerabilidades de SCA.')
      except Exception as e:
          message= f"<html><head></head><body><h2>No se encontraron vulnerabilidades de SCA.</h2></body></html>"
          with open('$(Build.ArtifactStagingDirectory)/reporte-osv-scanner.html', 'w') as f:
            f.write(message)
          print(f"Error: {e}")
    environmentVariable: 'PYTHONPATH'
  condition: always()

# Se publican los resultados de la prueba
- task: PublishHtmlReport@3
  displayName: 'Publicando reporte'
  inputs:
    tabName: 'Reporte OSV Scanner'
    reportDir: '$(Build.ArtifactStagingDirectory)/reporte-osv-scanner.html'
  condition: always()

# Publicación de reporte HTML
- task: PublishBuildArtifacts@1
  displayName: 'Publicando reporte HTML'
  inputs:
    ArtifactName: 'Reporte-OSV-Scanner(SCA)'
    PathtoPublish: '$(Build.ArtifactStagingDirectory)/reporte-osv-scanner.html'
  condition: always()

# Limpieza de los archivos temporales de ArtifactStagingDirectory
- task: DeleteFiles@1
  displayName: 'Limpiando ArtifactStagingDirectory'
  inputs:
    SourceFolder: '$(Build.ArtifactStagingDirectory)'
    Contents: |
      reporte-osv-scanner.json
    removeSourceFolder: false
    cleanTargetFolder: false
  condition: always()
```

Anexo C: Código para el cálculo de métricas de rendimiento

```
import pandas as pd
# Importar archivos de vulnerabilidades
df_benchmark = pd.read_csv('cwes_Benchmark.csv', sep = ';', encoding='utf-8')
df = pd.read_html('reporte-semgrep.html', encoding = 'utf-8')
df = df[0].set_index('Regla de Semgrep')

# Limpieza de índices
df = df.loc[:, ~df.columns.str.contains('^Unnamed')]
# Agregar columnas de identificadores CWE
cwes= df['CWE'].str.extract(r'(CWE-\d+)', expand=False)
df['CWE'] = cwes
# Agregar nombre del archivo que contiene el test de Benchmark
df['Archivo'] = df['Archivo afectado'].str.extract(r'(BenchmarkTest\d+)', expand=False)
# Correción de valores nulos
df.dropna(subset=['Archivo'], inplace=True)
# Exportar csv para visualizacion
```

```
df.to_csv('semgrep_prueba.csv')

# Agrupar por CWES
df['Archivo'] = df['Archivo afectado'].str.extract(r'(BenchmarkTest\d+)', expand=False)
df_cwes_archivos =df.groupby('Archivo')
df_cwes_agr =df.groupby('CWE')
verdadero_positivo = 0
verdadero_negativo= 0
falso_negativo  = 0
falso_positivo = 0

# Recorrer ambos archivos y comparar CWEs
for x, y in df_cwes_archivos:
    for i in df_benchmark.iterrows():
        if (i[1]['test name'] == x and (i[1]['cwe'] != 'CWE-327')):
            lista_cwes = list(y['CWE'])
            if (i[1]['cwe'] in lista_cwes) and i[1]['real vulnerability'] == True:
                verdadero_positivo += 1
                longitud_cwes = len(lista_cwes) - 1
                falso_positivo += longitud_cwes
            elif (i[1]['cwe'] in lista_cwes) and i[1]['real vulnerability'] == False:
                falso_positivo += len(lista_cwes)
            elif (i[1]['cwe'] not in lista_cwes) and i[1]['real vulnerability'] == True:
                falso_negativo += 1
                falso_positivo += len(lista_cwes)
            elif (i[1]['cwe'] not in lista_cwes) and i[1]['real vulnerability'] == False:
                verdadero_negativo += 1
                falso_positivo += len(lista_cwes)
            break
        elif (i[1]['test name'] == x) and (i[1]['cwe'] == 'CWE-327'):
            lista_cwes = list(y['CWE'])
            lista_cwes = list(set(lista_cwes))
            if ('CWE-326' in lista_cwes) and i[1]['real vulnerability'] == True:
                verdadero_positivo += 1
                longitud_cwes = len(lista_cwes) - 1
                falso_positivo += longitud_cwes
            elif ('CWE-326' in lista_cwes) and i[1]['real vulnerability'] == False:
                falso_positivo += len(lista_cwes)
            elif ('CWE-326' not in lista_cwes) and i[1]['real vulnerability'] == True:
                falso_negativo += 1
                falso_positivo += len(lista_cwes)
            elif ('CWE-326' not in lista_cwes) and i[1]['real vulnerability'] == False:
                verdadero_negativo += 1
                falso_positivo += len(lista_cwes)
            break
print(f'Verdaderos positivos: {verdadero_positivo}')
print(f'Verdaderos negativos: {verdadero_negativo}')
print(f'Falsos negativos: {falso_negativo}')
print(f'Falsos positivos: {falso_positivo}')
```

Anexo D: Código para ejecución de pruebas de seguridad con herramientas seleccionadas

En esta sección se trabaja con repositorio master-data-api-oke del Grupo Éxito.

```
resources:
  repositories:
  - repository: self
    name: $(System.TeamProject)/$(Build.Repository.Name)
    type: git
    ref: $(Build.SourceBranch)

jobs:
#  Semgrep
#**********************************
```

```
- job: SemgrepOSS_SAST
  displayName: 'Semgrep(SAST)'
  pool:
    name: 'Azure Pipelines'
    vmImage: 'ubuntu-latest'

  steps:
  - checkout: self
    fetchDepth: 1

  # Instalando Semgrep
  - task: Bash@3
    displayName: 'Instalando Semgrep'
    inputs:
      targetType: 'inline'
      bash: docker pull semgrep/semgrep:latest

  # Ejecutando Semgrep xml
  - task: Bash@3
    displayName: "Ejecutando Semgrep"
    inputs:
      targetType: 'inline'
      script: |
        docker run --rm \
        -v "$(System.DefaultWorkingDirectory):/src:cached" \
        -v "$(Build.ArtifactStagingDirectory):/reports:cached" \
        semgrep/semgrep:latest \
        semgrep scan \
        --config "p/r2c-security-audit" \
        --include /src \
        --force-color \
        --metrics=off \
        --pro \
        -q \
        --json-output /reports/reporte-semgrep.json
        report_json="$(Build.ArtifactStagingDirectory)/reporte-semgrep.json"

        # Verificación del formato del reporte
        if [ ! jq . "$report_json" > /dev/null 2>&1 ]; then
          echo "NO se encontro reporte en formato JSON o el reporte no se formateó correctamente."
          exit 1
        fi

        # Manejo de errores para jq
        jq_validation() {
            jq "$@" "$report_json" || {
                echo "Escaneo de JSON Fallo. Revisar jerarquía y nombre de variables."
                exit 1
            }
        }
        severity_warning=$(jq_validation '[.results[]?
            | select(.severity == "WARNING" and (.extra.impact == "HIGH" and
            (.extra.likelihood == "MEDIUM" or .extra.likelihood == "HIGH" )))] | length')
        results_len=$(jq_validation '.results | length')

        # Ruptura del pipeline ante Severidad ERROR
        if grep -q '"severity": "ERROR"' "$report_json"; then
          echo "Se hallaron vulnerabilidades con severidad ERROR."
          echo "Por favor revise el reporte de los hallazgos que le ayudará a identificarlas y solucionarlas."
          exit 1
        elif [ "$results_len" -eq 0 ]; then
            echo "No se encontraron vulnerabilidades."
        else
          echo "No se hallaron vulnerabilidades con Severidad ERROR"
        fi
```

```
    # Ruptura del pipeline ante severidad WARNING con Impacto High
    if [ "$severity_warning" -gt 0 ]; then
      echo "Se encontraron vulnerabilidades con severidad WARNING e impacto alto."
      echo "Por favor revise el reporte de los hallazgos que le ayudará a identificarlas y solucionarlas."
      exit 1
    elif [ "$results_len" -eq 0 ]; then
       echo "No se encontraron vulnerabilidades."
    else
      echo "No se encontraron vulnerabilidades con severidad WARNING e impacto alto"
    fi
  continueOnError: "false"

 # Instalación de dependencias
- task: Bash@3
  displayName: 'Instalación de dependencias'
  inputs:
    targetType: 'inline'
    script: pip install --upgrade pandas jinja2
  condition: always()

# Generación de reporte
- task: PythonScript@0
  displayName: 'Generando reporte'
  inputs:
    scriptSource: 'inline'
    script: |
      import json
      import pandas as pd
      from urllib.parse import quote
      try:
          with open('$(Build.ArtifactStagingDirectory)/reporte-semgrep.json', 'r') as d:
            findings = json.load(d)
          if 'results' in findings and len(findings['results']) >0:
            data = []
            for finding in findings['results']:
                check_id = finding.get('check_id', '')

                # Se muestra la ruta del archivo en varias lineas
                path = finding.get('path', '')
                long = len(path)
                ind_div1 = path.find('/', long // 4)
                ind_div2 = path.find('/', 2 * long // 4)
                ind_div3 = path.find('/', 3 * long // 4)
                if ind_div1 != -1 and ind_div2 != -1 and ind_div3 != -1:
                    path = path[:ind_div1+1] + '<br>' + path[ind_div1+1:ind_div2+1] + '<br>' + path[ind_div2+1:ind_div3+1] +
                            '<br>' + path[ind_div3+1:]
                elif ind_div1 != -1 and ind_div2 != -1:
                    path = path[:ind_div1+1] + '<br>' + path[ind_div1+1:ind_div2+1] + '<br>' + path[ind_div2+1:]
                elif ind_div1 != -1:
                    path = path[:ind_div1+1] + '<br>' + path[ind_div1+1:]
                else:
                    path = path
                # Se define el contenido de las variables que se mostrarán en la tabla del HTML
                start_line = finding.get('start', {}).get('line', '')
                description = finding.get('extra', {}).get('message', '')
                impact = finding.get('extra', {}).get('metadata', {}).get('impact', '')
                category = finding.get('extra', {}).get('metadata', {}).get('category', '')
                cwe_value = finding.get('extra', {}).get('metadata', {}).get('cwe', '')
                if isinstance(cwe_value, list):
                    cwe = ', '.join(cwe_value)
                else:
                    cwe = cwe_value
                likelihood = finding.get('extra', {}).get('metadata', {}).get('likelihood', '')
                vulnerability_class = ', '.join(finding.get('extra', {}).get('metadata', {}).get('vulnerability_class', ''))
```

```
    technology = ', '.join(finding.get('extra', {}).get('metadata', {}).get('technology', ''))
    severity = finding.get('extra', {}).get('severity', '')
    references = ', '.join(finding.get('extra', {}).get('metadata', {}).get('references', ''))

    # Se muestra la url de la regla de Semgrep en varias lineas
    url = "https://semgrep.dev/r/" + quote(check_id)
    long_url = len(url)
    url_div1 = url.find('/', long_url // 4)
    url_div2 = url.find('/', 2 * long_url // 4)
    url_div3 = url.find('/', 3 * long_url // 4)
    if url_div1 != -1 and url_div2 != -1 and url_div3 != -1:
        url_mult = url[:url_div1+1] + '<br>' + url[url_div1+1:url_div2+1] + '<br>' + url[url_div2+1:url_div3+1] +
                   '<br>' + url[url_div3+1:]
    elif url_div1 != -1 and url_div2 != -1:
        url_mult = url[:url_div1+1] + '<br>' + url[url_div1+1:url_div2+1] + '<br>' + url[url_div2+1:]
    elif url_div1 != -1:
        url_mult = url[:url_div1+1] + '<br>' + url[url_div1+1:]
    else:
        url_mult = url
    # Se especifican las variables y la información que se mostrarán en el HTML
    data.append({
        'Semgrep Rule': f"<a href={url} target='_blank'>{url_mult}</a>",
        'Severity': severity,
        'Vulnerability Class': f"{technology}: {vulnerability_class} Start line - {start_line}",
        'File': path
    })
df = pd.DataFrame(data)
# Se define el estilo que tendrá la tabla en el archivo HTML
styles = [
    dict(selector="tr:hover", props=[("background-color", "#121614")]),
    dict(selector="table", props=[("margin", "auto"), ("width", "100%"), ("font-family", "Arial, sans-serif"),
    ("background-color", "#334139")]),
    dict(selector="th", props=[("font-size", "14px"), ("background-color", "#007bff"), ("color", "white"), ("padding",
    "12px"), ("text-align", "center")]),
    dict(selector="td", props=[("overflow-wrap", "break-word"), ("font-size", "12px"), ("text-align", "left"),
    ("padding", "8px")])]
df.index = df.index + 1
html_table = (df.style
        .set_table_styles(styles).to_html(index=False, escape=False))
html_page = f"""
<!DOCTYPE html>
<html>
<head>
    <title>Reporte Semgrep</title>
    # Se define el estilo para la página HTML
    <style>
        body {{
            background-color: #000000;
            color: #ffffff;
            font-family: "Overpass Mono", monospace;
        }}
        h1 {{
            text-align: center;
            color: #ffffff;
            padding: 20px;
            font-size: 2em;
            border-bottom: 1px solid #ffffff;
        }}
        table {{
            margin: auto;
            border-collapse: collapse;
            width: 80%;
            font-size: 16px;
            color: #ffffff;
        }}
```

```
                th, td {{
                    padding: 8px;
                }}
                tr:hover, td:hover {{
                    background-color: #090B0A;
                }}
                th {{
                    padding-top: 12px;
                    padding-bottom: 12px;
                    text-align: left;
                    background-color: #4CAF50;
                    color: white;
                }}
            </style>
            </head>
            <body>
                <h1>Reporte Semgrep</h1>
                {html_table}
            </body>
            </html>
            """
            # Se guarda el archivo HTML con estilo CSS aplicado
            with open('$(Build.ArtifactStagingDirectory)/reporte-semgrep.html', 'w') as d:
                d.write(html_page)
          else:
            message= f"<html><head><body></html><h2>No se encontraron vulnerabilidades SAST.</h2></head></body>"
            with open('$(Build.ArtifactStagingDirectory)/reporte-semgrep.html', 'w') as f:
              f.write(message)
            print('No se encontraron vulnerabilidades SAST.')
      except Exception as e:
          print(f"Error: {e}")
    environmentVariable: 'PYTHONPATH'
  condition: always()
 # Publicando resultados de la prueba
 - task: PublishHtmlReport@3
  displayName: 'Publicando reporte'
  inputs:
    tabName: 'Reporte Semgrep'
    reportDir: '$(Build.ArtifactStagingDirectory)/reporte-semgrep.html'
  condition: always()

 # Limpieza de los archivos temporales de ArtifactStagingDirectory
 - task: DeleteFiles@1
  displayName: 'Limpiando ArtifactStagingDirectory'
  inputs:
    SourceFolder: '$(Build.ArtifactStagingDirectory)'
    Contents: |
      reporte-semgrep.json
    removeSourceFolder: false
    cleanTargetFolder: false
  condition: always()

#  Trivy
# #*********************************
- job: 'Trivy_SCA_IAC'
  displayName: 'Trivy(SCA_IAC)'
  pool:
    name: 'Azure Pipelines'
    VmImage: Ubuntu-latest
  steps:
    - checkout: self
      fetchDepth: 1

    # Carga de vulnerabilidades sin parche y/o sin remediación.
    - task: Bash@3
```

```
  displayName: 'Cargando archivo .trivyignore'
  inputs:
    targetType: 'inline'
    script: |
      cat << EOF > .trivyignore
      AVD-DS-0002 exp:2024-06-02
      EOF
# Configuración de los parámetros de escaneo Trivy
- task: trivy@1
  displayName: 'Ejecutando Trivy'
  inputs:
    options: --scanners "vuln,misconfig,secret,license" --exit-code 1 --severity "CRITICAL,HIGH" --ignorefile
             .trivyignore
    version: latest
    ignoreUnfixed: true
    path: .
  continueOnError: false
```