



TRABAJO FIN DE MÁSTER

MÁSTER EN CIBERSEGURIDAD Y PRIVACIDAD

DevSecOps en aplicaciones Docker y/o Kubernetes

Abraham Santana Cebrián

Tutor: Micael Gallego Carrillo

CONTENIDO

1	Introducción y Contexto	5
1.1	Tema del Trabajo	5
1.2	Motivación y justificación	6
2	Organización	8
2.1	Objetivos	8
2.2	Metodología	10
2.3	Planificación del Trabajo	11
3	Estado del Arte	12
3.1	Docker	12
3.2	Integración Continua	13
3.2.1	Jenkins	14
3.3	Static Code Analysis y Static Application Security Testing	15
3.3.1	Reglas del SCA y SAST	16
3.3.2	Niveles de Severidad	17
3.3.3	Modos de ejecución	19
3.4	Static Code Analysis en Ciberseguridad: SAST	20
3.4.1	SonarQube	21
3.4.2	OWASP Dependency Check	21
3.4.3	Clair: Static Analysis of Vulnerabilities	22
3.5	Dynamic Application Security Testing (DAST)	25
3.5.1	OWASP ZAP	25
3.6	Aplicación Web Vulnerable: WebGoat	26
3.6.1	WebGoat	26
3.7	Kubernetes	27
3.7.1	Kompose	27
4	Desarrollo del Proyecto	28
4.1	Análisis	28
4.2	Diseño del Ciclo de Vida del Software DevSecOps	28
4.2.1	Etapas DevOps	29
4.2.2	Etapas DevSecOps	30
4.3	Implantación	32
4.3.1	Docker-Compose: orquestación del entorno DevSecOps	32
4.3.2	Jenkins: configuración y puesta a punto	35
4.3.3	SonarQube: configuración y puesta a punto	41
4.4	Codificación y Pruebas	43
4.4.1	Configuración del Pipeline	44
4.4.2	Etapas del Pipeline	45

4.4.3	Migración a Kubernetes con Kompose	48
5	Resultados Obtenidos	49
5.1	Entorno Docker	49
5.1.1	Declarative: Tool Install	49
5.1.2	Checkout repository changes	49
5.1.3	SAST analysis	50
5.1.4	Dependency analysis	53
5.1.5	Release and Docker Build	54
5.1.6	Docker Security scanner	55
5.1.7	Docker Deployment	57
5.1.8	DAST Analysis	57
5.1.9	Stop and Delete App Docker Container	58
5.2	Entorno Kubernetes	59
6	Dificultades encontradas	60
6.1	Docker	60
6.2	WebGoat	60
6.3	Clair y Clairctl	60
6.4	Herramientas y Despliegues no realizados	61
6.4.1	Kompose	61
6.4.2	Notary	61
7	Conclusiones	62
8	Líneas de trabajos futuros	64
9	Bibliografía y Referencias	65
10	Tabla de Ilustraciones	70
11	Anexo 1: Docker Compose File script	72
12	Anexo 2: Pipeline script	73

1 INTRODUCCIÓN Y CONTEXTO

DevOps es un término que en inglés proviene de la composición de dos términos: Development y Operations; esto es, Desarrollo y Operaciones. Es una práctica de ingeniería de software que tiene como objetivo unificar el desarrollo de software (Dev) y la operación del software (Ops). La principal característica del movimiento DevOps es la de defender enérgicamente la automatización y la monitorización en todos los pasos del desarrollo del software, desde la integración, las pruebas, la liberación hasta la implementación y la administración de la infraestructura [1].

DevSecOps consiste en añadirle seguridad desde el principio al concepto de DevOps, haciendo que todo el mundo esté concienciado e implicado en que todo el proceso de creación del software debe ser seguro; es decir, haciéndoles responsables de la seguridad con el objetivo de implementar decisiones y acciones de seguridad a la misma escala y velocidad que las de desarrollo (Dev) y las de operación (Ops).

La principal diferencia que tiene DevSecOps con su antecesor DevOps es el cambio en el punto de vista de la integración de la seguridad. Este último tradicionalmente ha dejado para el final todo lo relacionado con la seguridad, en cambio, DevSecOps trata de intentar automatizar las tareas de seguridad principales al incorporar controles y procesos de seguridad desde el principio del flujo de trabajo [2].

1.1 TEMA DEL TRABAJO

El objetivo fundamental de este Trabajo Final de Máster (en adelante, Trabajo) es la continuación del Trabajo Final de Máster (en adelante, TFM) que he realizado para el Máster de Software Craftsmanship de la Universidad Politécnica de Madrid (UPM) en el cual se realizó la construcción, el estudio e implementación de un Pipeline de DevSecOps en un entorno de Integración Continua (CI) y Despliegue Continuo (CD) en el cual serán implementadas todas las herramientas de seguridad posibles.

En aquel TFM, se utilizó la Forja proporcionada por CodeURJC [3], ya que en ella se despliega un entorno real de integración continua, teniendo las herramientas necesarias para su realización en la vida real, las cuales son Jenkins como un servidor de CI, Gerrit como repositorio de código, un servidor LDAP para la gestión de roles, un servidor web Apache, etc. A partir de los resultados obtenidos durante el desarrollo de este Trabajo, se realiza una demo [4] sobre la configuración adecuada para que sean seguras todas las partes implicadas en el proyecto.

Como se puede comprobar consultando dicho TFM preliminar, los resultados obtenidos utilizando la forja anterior estuvieron condicionados a la limitación de memoria RAM en la máquina física donde se pretendía virtualizar todo el entorno. Por ello, se ha decidido continuar el trabajo en el punto dejado para alcanzar los objetivos fijados, tomando la decisión de migrar todo el ecosistema al proveedor de servicios en la nube Amazon Web Services (AWS) [5] la cual permite utilizar unos recursos escalables en función de las necesidades del cliente, que en este caso son mayores de los que puede proporcionar una visualización de un host con Ubuntu con una memoria RAM de 13GB dedicados.

En consecuencia, en este Trabajo, se migrará el entorno de Integración Continua / Despliegue Continuo (CI/CD) al proveedor AWS. El objetivo es desacoplar el entorno CI/CD de una máquina física o virtualizada, ofreciendo las herramientas del entorno, para poder configurar y desplegar en AWS dicho ecosistema. Esto hará que sea mucho más potente que un simple Host en el que desplegar directamente o virtualizar todas las herramientas necesarias para el desarrollo del Pipeline DevSecOps.

1.2 MOTIVACIÓN Y JUSTIFICACIÓN

Hoy en día vivimos en un mundo VUCA (*Volatility, Uncertainty, Complexity, and Ambiguity*, en español sería *Volatilidad, Incertidumbre, Complejidad, y Ambigüedad*) [6], desde las infraestructuras y las tecnologías, hasta las necesidades de las empresas y los clientes, haciendo que todos esos cambios afecten al desarrollo de los proyectos software. Por ello, las metodologías en el desarrollo software han ido cambiando y evolucionando para adaptarse a ese vertiginoso ritmo actual [7] [8]:

- **Cascada:** metodología secuencial (la tradicional) la cual se divide en etapas en las que se requisa todo, se analiza todo, se diseña todo, se programa todo (incluidas las pruebas), se prueba todo, y se entrega. El problema de esta metodología es que si sucede algún fallo durante algunas de las etapas se vuelve a la primera, volviendo a empezar todas las etapas [9].

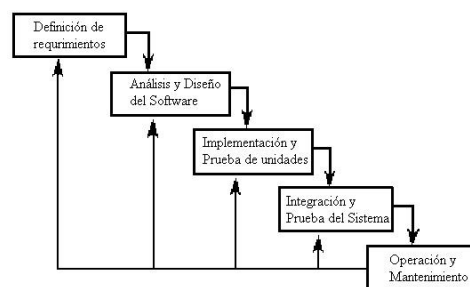


Ilustración 1 Modelo cascada realimentado para el ciclo de vida [9]

- **Iterativas (agile):** está basada en la metodología cascada y en sus etapas, pero “troceando” dichas etapas en otras más pequeñas, como porciones, que se irán completando en cada iteración completa que se realice. Es decir, en cada iteración se requisa un poco, se analiza un poco, se diseña un poco, se programa un poco (incluidas pruebas), y se despliega un poco [1] [10].

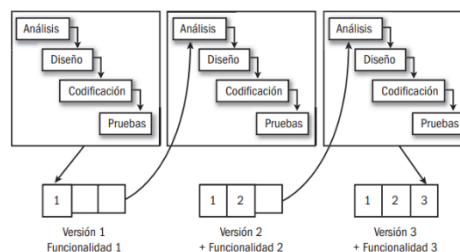


Ilustración 2 Modelo iterativo incremental para el ciclo de vida [80]

DevOps surge por el gran éxito de las metodologías ágiles del desarrollo del software. Las empresas y organizaciones querían lanzar su software con una frecuencia mayor, querían que fuera más rápido el proceso de entrega del software. Por ello surgió DevOps, solucionando la tensión que suponía para la gestión de versiones, la integración continua, y la entrega continua, creando patrones de automatización de cada una de ellas [1].

DevSecOps se crea a partir del concepto general y de los valores de DevOps, teniendo como objetivo la integración de la seguridad en todos los pasos del proceso de desarrollo, haciendo que la seguridad se integre en el desarrollo de manera activa [11].



Veracode realiza un informe anual sobre el estado de la seguridad de software. El último es el décimo, del 2019, han evaluado 1,4 millones de escaneos realizados a 85 mil aplicaciones durante todo el año, obteniendo que el 83% de las aplicaciones tienen fallos de seguridad en los escaneos iniciales que se les ha realizado. Además, proporciona información sobre los tipos más comunes de vulnerabilidades y las prácticas que ofrecen los mejores resultados y rendimiento en la industria. El informe encontró que la mayoría de los fallos se corrigen (56%) y que las empresas que realizan análisis y escaneos con mayor frecuencia tienen alrededor de 5 veces menos deuda técnica en seguridad que aquellas empresas que realizan análisis más livianos. Esto es debido a que tienen pruebas de seguridad automatizadas, haciendo que las actividades de seguridad sean algo habitual, haciendo que todo el mundo esté concienciado e implicado en que todo el proceso de creación del software debe ser seguro, es decir, haciéndoles responsables de la seguridad con el objetivo de implementar decisiones y acciones de seguridad a la misma escala y velocidad que las de desarrollo (Dev) y las de operación (Ops). En definitiva, el informe sugiere que las prácticas DevSecOps mejoran la seguridad general del software [12].

Estos son los motivos fundamentales para estudiar, montar, e implementar un Pipeline DevSecOps, implementado todas las herramientas de seguridad posibles, entre otras, del tipo análisis estático del código y dependencias (ya que es básico para el desarrollo de proyectos software que dicha calidad sea mínima), análisis dinámico de código, criterios de análisis de la seguridad, análisis de vulnerabilidades Docker, *pen testing*, etc. Todo ello asumiendo que el proyecto de desarrollo software sobre el que implementar el Pipeline DevSecOps es una aplicación web ya desarrollada, puesto que no es objeto de este Trabajo.

2 ORGANIZACIÓN

2.1 OBJETIVOS

El objetivo fundamental de este Trabajo es el estudio, montaje, e implementación de un Pipeline de DevSecOps, integrando en la implementación todas las herramientas de seguridad posibles, asumiendo que el proyecto de desarrollo software es una aplicación web ya desarrollada (no es motivo de este Trabajo el desarrollo del producto software).

Se desea implementar herramientas para todas las etapas DevSecOps, ya que la aplicación web será posteriormente empaquetada en una imagen Docker, a la cual le serán aplicados estrategias de seguridad específicas, y finalmente será desplegada para aplicar estrategias de seguridad específicas en la etapa de despliegue. Las herramientas que pueden ser implementadas pueden ser del tipo:

- Análisis Estático del Código y Dependencias (SonarQube y OWASP Dependency Check)
- Análisis de Vulnerabilidades Docker (Clair: Static Analysis of Vulnerabilities)
- Análisis Dinámico de Código (OWASP ZAP)
- Pen Testing (OWASP ZAP)

En el caso de que haya tiempo en el desarrollo de este proyecto, uno de los objetivos será la realización del mismo procedimiento para un entorno en Kubernetes, replicando lo conseguido en un entorno Dockers.

El objetivo principal es tener una visión global de las herramientas, de las tecnologías utilizadas, y del sistema de integración de los elementos, usando como base artículos, posts, informaciones, papers, otros trabajos, etc., en relación con la materia.

Se materializará con una infraestructura configurada a modo de tutorial, explicando cómo se monta un entorno de Continuous Integration / Continuous Deployment (CI/CD), y la configuración de las herramientas utilizadas en este proyecto. Todo ello se documenta de la forma más exhaustiva posible.

El siguiente objetivo está relacionado con la asignatura de Software seguro: con la base explicada en la asignatura, se indaga en aplicaciones específicas a partir de herramientas concretas, poniéndolas a funcionar de manera definida, teniendo como objetivo final llegar hasta un escenario con la tecnología de Kubernetes.

A continuación, se sintetizan los objetivos marcados para este Trabajo Fin de Máster:

- Diseño de un ciclo de vida del desarrollo software seguro.
- Diseño e implementación de un Sistema *Continuous Integration (CI) / Continuous Deployment (CD)*, es decir, CI/CD, en un entorno Docker, al cual se le implementarán las diferentes herramientas durante el flujo del Pipeline DevSecOps. Basado en servidor CI/CD.
- Diferenciación de las distintas etapas del Pipeline DevSecOps para la constitución de la seguridad durante el ciclo de vida del desarrollo software, y automatización de todas ellas.
- Búsqueda e implementación de herramientas para obtener las siguientes funcionalidades de seguridad:
 - Análisis y definición de políticas y requisitos de seguridad que debe cumplir el producto software, en este caso, una aplicación web ya desarrollada.
 - Análisis de Código Estático de la aplicación durante el proceso de Integración Continua (SonarQube).
 - Análisis de Dependencias de la aplicación durante el proceso de Integración Continua (OWASP Dependency Check).
 - Análisis Estático de Vulnerabilidades en imágenes Docker generadas durante el proceso de Despliegue Continuo (Clair).
 - Análisis de Código Dinámico y auditoría durante el despliegue de la aplicación web como etapa final DevSecOps (OWASP ZAP).
 - Notary para analizar y asegurar que las imágenes estén firmadas.
- Seleccionar una aplicación web vulnerable ya desarrollada como base para la implementación, desarrollo, y simulación, de todo el ecosistema CI/CD, en consecuencia, de su ciclo de vida de desarrollo software (WebGoat).
- Finalmente, se realizará una migración y adaptación de todo el sistema a un entorno en Kubernetes.

Una vez se obtengan los resultados buscados, se realizará una demo en vídeo para materializar la configuración adecuada y su funcionamiento para que sean seguras todas las partes implicadas en el Pipeline, mostrando el funcionamiento y resultados de un Pipeline DevSecOps.

2.2 METODOLOGÍA

La metodología utilizada para este Trabajo se ha basado en la metodología científica [13].

Se ha partido de la base del resultado obtenido en TFM realizado anteriormente, en el que se pudo obtener el resultado de que es limitante el uso de un Host con una cantidad de memoria RAM escasa para el uso de todo el ecosistema desplegado en Docker

La metodología empleada en este Trabajo Final de Máster ha sido la siguiente:

- Se ha realizado una revisión bibliográfica de las publicaciones disponibles en internet acerca de del estado del arte de las metodologías del desarrollo software, centrándose en las metodologías iterativas, a la que pertenece DevOps de la que nace la metodología DevSecOps. Se ha hecho uso, entre otros, de los siguientes buscadores:
 - <https://brain.urjc.es/>
 - <https://www.elsevier.com/es-es>
 - <https://scholar.google.es/>
 - <https://github.com/>
- Se ha seleccionado una aplicación web desarrollada en Docker y/o Kubernetes, ya que no es motivo de este proyecto desplegar la misma en esa tecnología, para así desarrollar el mencionado DevSecOps Pipeline.
- Se ha realizado un análisis de las técnicas utilizadas para la implementación de la seguridad en el desarrollo software, dando como resultado las herramientas utilizadas en este trabajo.
- En consecuencia, se ha realizado una investigación de las fuentes primarias de dichas herramientas tanto para su documentación como para su implementación en el Pipeline DevSecOps.
- Realizar la integración progresiva de las herramientas en el Pipeline modularizando el proceso:
 - Instalación y despliegue de la herramienta.
 - Pruebas de su integración en el sistema.
 - Análisis de los resultados y comportamiento.
 - Pruebas de regresión para comprobar la integridad, estabilidad, y rendimiento del ecosistema como sistema complejo integrado.
- Por último, ha podido confirmar que el uso de un entorno sin limitaciones físicas para poder desplegar todo el ecosistema elaborado en Dockers ha permitido conseguir la mayoría de los objetivos planteados para este Trabajo.

2.3 PLANIFICACIÓN DEL TRABAJO

Se ha realizado la siguiente planificación general para la finalización de este Trabajo:

Tasks				
WBS	Name	Start	Finish	Work
1	Revisión bibliográfica del Estado del Arte	Jun 1	Jun 9	10d
1.1	Búsqueda y selección de app vulnerable	Jun 1	Jun 3	3d
1.2	Búsqueda de material	Jun 1	Jun 9	7d
2	Diseño de un Ciclo de Vida del Desarrollo Software Seguro	Jun 10	Jun 22	16d 2h
2.1	Análisis de Técnicas encontradas / selección de Herramientas	Jun 10	Jun 22	8d 1h
2.2	Documentación de Fuentes Primarias	Jun 10	Jun 22	8d 1h
3	Implementación escosistema DevSecOps	Jun 22	Sep 4	143d
3.1	Codificación y Pruebas	Jun 22	Sep 4	54d
3.2	Despliegue de Jenkins	Jun 22	Jun 25	3d
3.3	Despliegue de SonarQube	Jun 25	Jun 30	3d
3.4	Integración de Docker en Jenkins	Jun 30	Jul 20	14d
3.5	Dificultades con Docker en Jenkins	Jun 30	Jul 16	12d
3.6	Integración de Clair	Jul 20	Aug 3	10d
3.7	Dificultades con Clair	Jul 20	Jul 30	8d
3.8	Integración de OWASP Dependency-Check	Aug 3	Aug 12	7d
3.9	Integración de OWASP ZAP	Aug 12	Sep 1	14d
3.10	Dificultades con OWASP ZAP	Aug 12	Aug 28	12d
3.11	Migración a Kubernetes	Sep 1	Sep 4	3d
3.12	Dificultades con Kubernetes	Sep 1	Sep 4	3d
4	Redacción del Documento de Predefensa	Jun 1	Jul 8	28d
5	Presentación de Predefensa	Jul 6	Jul 7	2d
6	Redacción del Documento Final	Jul 9	Sep 4	42d
7	Video de Demostración de Defensa	Sep 2	Sep 2	1d
8	Presentación de Defensa	Sep 3	Sep 4	2d

Ilustración 3 Tareas planificadas

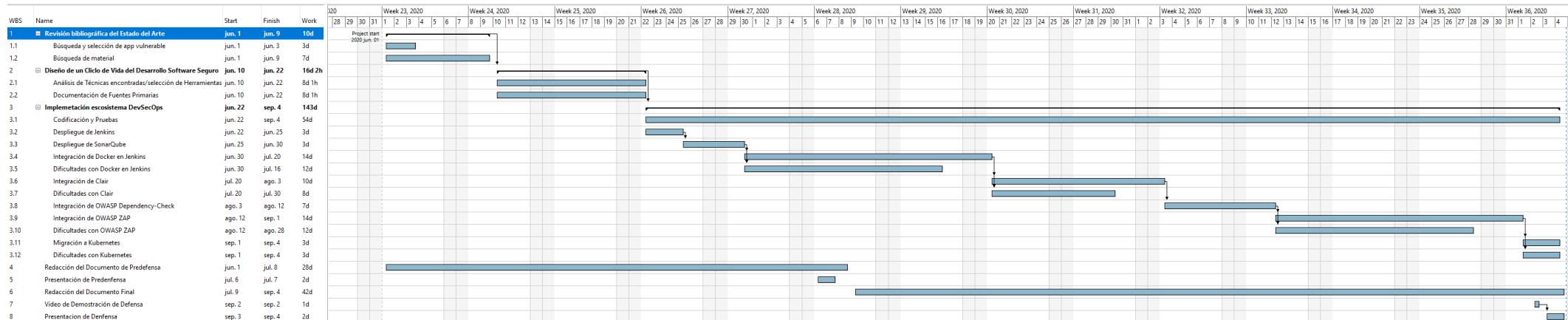


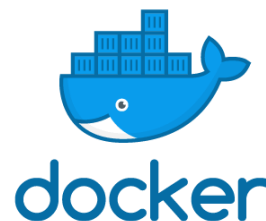
Ilustración 4 Planificación con el Diagrama de Gantt

3 ESTADO DEL ARTE

Se pretende mostrar cuál es el estado del arte en relación con las herramientas disponibles, las usadas y su relación con DevSecOps.

3.1 DOCKER

Docker es un proyecto *open-source* que permite desplegar aplicaciones dentro de contenedores, proporcionando a cada una de ellas un gran nivel de abstracción e independencia respecto de las otras. Los contenedores tardan milisegundos en arrancar y consumen únicamente la memoria que necesita la aplicación ejecutada en el contenedor. Esto hace que no necesite reservar la memoria completa como lo haría una Máquina Virtual (VM). Por otro lado, también permite exportar cada contenedor con su aplicación de una máquina a otra, ahorrando tiempo en instalaciones [14].



Inicialmente desarrollada para Linux, aunque dispone de herramientas para desarrolladores en Windows y MAC.

Para ejecutar un contenedor no se necesita *Hypervisor* porque no se ejecuta un sistema operativo invitado y no hay que simular *hardware*. El contenedor es ejecutado directamente por el *Kernel* del host como si fuera una aplicación más, pero de forma aislada del resto [14].

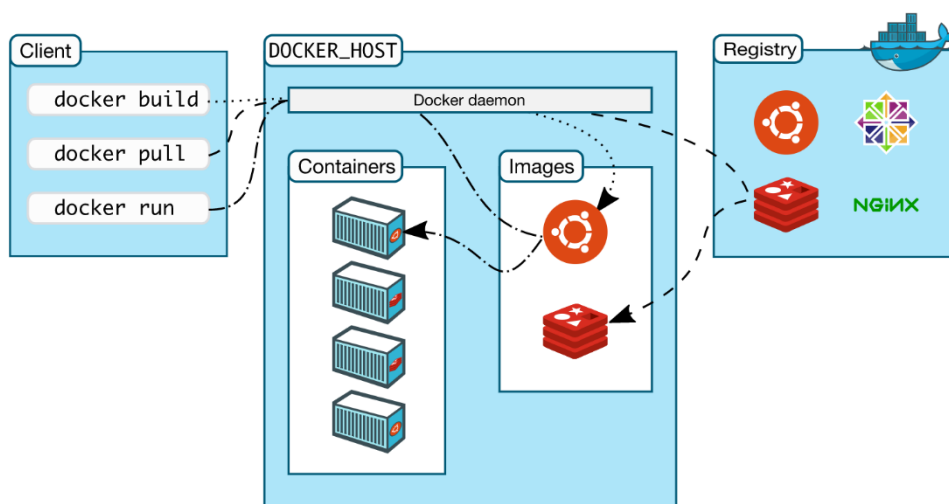


Ilustración 5. Esquema conceptual del funcionamiento de Docker [15] [16]

Para desplegar las aplicaciones en los contenedores, existe *DockerHub*, que es un repositorio centralizado donde se encuentran todas las imágenes de las distintas aplicaciones.

Las imágenes *Docker* se crean a partir de una plantilla, la cual incluye las herramientas básicas del sistema operativo de la plantilla (como pueden ser *Ubuntu* o *Alpine* entre otros), las librerías que necesita (como Java), y por supuesto, la aplicación que ha sido dockerizada. En el caso de que se quiera arrancar una imagen Docker que no está descargada localmente, se descargará automáticamente desde internet [14].

Para este Trabajo se ha elegido un ecosistema dockerizado que permite desacoplar la infraestructura local de la infraestructura del entorno DevSecOps buscado, ya que, al desarrollar el Trabajo en Docker, se da la posibilidad de usar el hardware que se quiera, simplemente hace falta tener instalado Docker allá donde se quiera montar el ecosistema y levantarlo.

3.2 INTEGRACIÓN CONTINUA

La integración continua (*Continuous Integration*, CI) surgió como consecuencia de la dificultad de integrar el trabajo de desarrollo de dos o más personas que han estado trabajando sobre la misma base de código. Este problema se le denominó como *Integration hell* al principio de practicarse la metodología de *eXtreme Programing* (Programación Extrema).

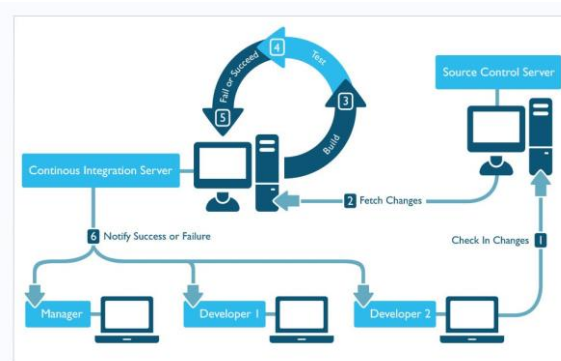


Ilustración 6 Esquema conceptual de Integración Continua y Despliegue Continuo [72]

CI es una práctica de desarrollo software donde los miembros del equipo integran su trabajo frecuentemente, como mínimo cada desarrollador o equipo de desarrollo realiza una vez al día la integración, aunque normalmente se realizan múltiples integraciones diarias.

Esto reduce significativamente los problemas de integración y permite desarrollar software de manera cohesiva y rápida. Para el correcto uso de la integración continua, y que ésta sea eficiente, se deben llevar a cabo determinadas prácticas, las siguientes son las mínimas que llevar a cabo [17]:

- Mantener un repositorio (rama) principal de integración: los proyectos de desarrollo de software involucran grandes cantidades de archivos que necesitan ser tratados como un conjunto. Para ayudar en esto, existen múltiples herramientas para este fin, de las cuales destaca Git.
- Automatizar los Builds de un proyecto: existen diferentes entornos para automatizar las *builds* de cualquier proyecto de desarrollo software. En este proyecto se utilizará Maven [18], por lo que haremos un brevemente comentario. Maven es una herramienta creada para realizar la *build* de un proyecto Java. Se basa en la utilización de un archivo llamado *Project Object Model* (POM), escrito en formato XML, donde se describe el proyecto a construir, cuáles son sus dependencias con otros módulos y componentes externos, y el orden en el que se deben construir los elementos. Tiene varias tareas ya definidas, como el empaquetado del proyecto (compilar todo el proyecto y crear un archivo .exe o .zip).
- Hacer que el proyecto ejecute sus propios tests: actualmente, los IDE de programación tienen una herramienta para hacer *debug* del código. Por ello, surgió la práctica de crear tests personalizados en la ejecución de la *build* del proyecto. Ello no significa que vayan a detectar el 100% de los fallos, pero si se puede llegar a detectar el 80% de ellos haciendo un esfuerzo de un 20% sobre el desarrollo del proyecto. Gracias a los conceptos

de *eXtreme Programming* (XP) y *Test Driven Development* (TDD), la práctica de incluir test personalizados a la hora de compilar el software se ha popularizado y los equipos de desarrollo se han dado cuenta de su utilidad.

- *Commit a la rama principal ejecuta una build*: cada vez que se realice una integración con la rama principal de desarrollo, siendo como mínimo uno al día, estas deben ejecutar los tests necesarios para garantizar la calidad de los *builds*, teniéndolos así "testeados". Esto significa que la rama principal estará siempre limpia y en un estado correcto, en teoría. En la práctica, los usuarios no suelen ejecutar una *build* manualmente por cada *commit* que realizan. Para solucionar esto, existen gestores de integración continua que permiten automatizar las *builds* de un proyecto software para su ejecución automáticamente después de cada *commit* realizado. Más adelante hablaremos sobre Jenkins, que es el usado en este proyecto.
- *Hacer pruebas en una copia del escenario de producción real*: la finalidad de realizar tests es detectar, bajo condiciones determinadas, cualquier problema que el proyecto pudiera tener en producción. Una condición que escapa a los tests en el entorno donde la aplicación va a ser ejecutada. Por ello es importante realizar pruebas en los distintos entornos donde la aplicación será ejecutada antes de llevarla a producción, aun cuando los tests no detecten fallos.

3.2.1 Jenkins

Jenkins es un servidor de integración continua, gratuito, *open-source* y actualmente uno de los más empleados para esta función, debido a su facilidad de uso [19].



Jenkins

Esta herramienta proviene de otra similar llamada Hudson, ideada por Kohsuke Kawaguchi, que trabajaba en Sun. Unos años después de que Oracle comprara Sun, la comunidad de Hudson decidió renombrar el proyecto a Jenkins, migrar el código a Github y continuar el trabajo desde ahí. No obstante, Oracle ha seguido desde entonces manteniendo y trabajando en Hudson. [20]

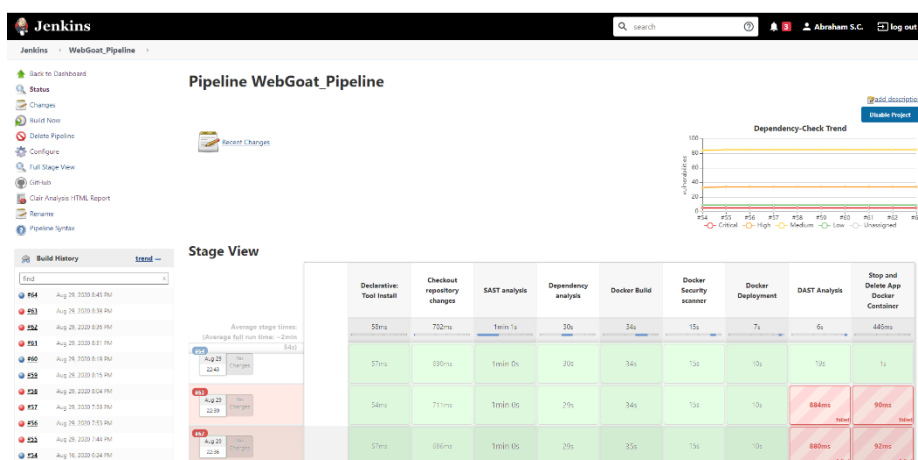


Ilustración 7. Visión general de un Pipeline en Jenkins

La base de Jenkins son las tareas, donde indicamos qué es lo que hay que hacer en un *Job*. Por

ejemplo, podríamos programar una tarea en la que se compruebe el repositorio de control de versiones cada cierto tiempo, y cuando un desarrollador quiera subir su código al control de versiones, este se compile y se ejecuten las pruebas.

Si el resultado no es el esperado o hay algún error, Jenkins notificará al desarrollador, al equipo de Aseguramiento de la Calidad (se usa con frecuencia el anglicismo *Quality Assurance*, QA), por email o cualquier otro medio, para que lo solucione. Si el *build* es correcto, podremos indicar a Jenkins que intente integrar el código y subirlo al repositorio de control de versiones.

Una de las cosas buenas que tiene Jenkins es que además de poder ayudarte a integrar el código periódicamente, puede actuar como herramienta que sirva de enlace en todo el proceso de desarrollo. Desde Jenkins se puede indicar que se lancen métricas de calidad y visualizar los resultados dentro de la misma herramienta. También podrás ver el resultado de los tests, generar y visualizar la documentación del proyecto, o incluso pasar una versión estable del software al entorno de QA para ser probado, a preproducción o producción.

Para trabajar con las distintas herramientas de desarrollo de software (*GitHub*, *BitBucket*, *Maven*, *Ant*, *Gradle*, etc.), así como añadir funcionalidades para obtener reportes de los tests o enviar notificaciones de correo cuando una *build* sea errónea, se utilizan diferentes *Plugins* añadidos a la configuración de Jenkins.

Para este Trabajo, se ha implementado esta herramienta como un servidor independiente desplegado en Docker [21] pero modificando la imagen para instalar en ella Docker y poder hacer uso de la herramienta [22].

3.3 STATIC CODE ANALYSIS Y STATIC APPLICATION SECURITY TESTING

El *Static Code Analysis* (SCA, en español sería Análisis Estático de Código), conocido también como *Static Program Analysis* (SPA, en español sería Análisis Estático de un Programa) [23], como su propio nombre indica, consiste en la realización de un análisis del código de un programa sin que éste sea ejecutado. En la mayoría de los casos, se realiza el análisis a través del código fuente, pero también cabe la posibilidad de realizar el análisis teniendo el código objeto (binario o *bytecode* en caso de lenguajes con máquinas virtuales).

SCA se utiliza cuando el análisis se realiza de forma automatizada por una herramienta. En cambio, si el análisis se realiza de manera manual, es decir, es una persona quien lo realiza, se emplea el término de *Code Review* (revisión de código).

Hay muchos niveles en los que puede realizarse el SCA, empezando por aquellas herramientas que únicamente detectan violaciones de *Indentation* (reglas de estilo, uso adecuado de formato de las variables, longitud máxima de las líneas...), hasta herramientas que realizan un análisis profundo del código fuente, pudiendo llegar a verificar formalmente desde si el programa se ajusta a una especificación dada, hasta si puede contener problemas de concurrencia.

En el ámbito de la seguridad del software, cuando el análisis de código se realiza con el objetivo de encontrar vulnerabilidades de seguridad, se suele usar el término *Static Application Security Testing* (SAST, el cual se desarrolla con más detalle en el punto 3.4 de este documento).

Existen multitud de páginas en la red¹ que listan diferentes herramientas de análisis estático de código [24].

3.3.1 Reglas del SCA y SAST

Las herramientas de análisis de código tienen un funcionamiento basado en la comprobación de ciertas reglas o patrones, existiendo diferentes tipos en función de que aspecto del código se está validando:

- Reglas de formato/estilo del código: son las más básicas. Verifican, entre otros, si los nombres de las variables están escritos de acuerdo con las reglas de estilo, si el tamaño máximo de línea de código, si las posiciones de las llaves de apertura y cierre de un bloque de código en las sentencias de control de flujo, etc. Las herramientas que comprueban únicamente este tipo de reglas suelen conocerse como *linters* [25].
- Uso/buenas prácticas de funcionalidades del lenguaje o librerías: este tipo de reglas se centran en aspectos de más alto nivel del código. Por ejemplo, en Java se considera buena práctica declarar las variables lo más cerca posible del lugar en el que se van a usar. Esto reduce el ámbito de las variables, lo que favorece la seguridad del código y su comprensión. Respecto a las librerías, existen buenas prácticas. Por ejemplo, leer el contenido de un fichero, se recomienda usar un buffer por eficiencia.
- Detección de Code Smell: más allá de las buenas prácticas en el uso del lenguaje y las librerías, subiendo de nivel de abstracción nos encontramos con la detección de malos olores (*code smell*). Se considera *bad smell* o *code smell* [26] una cierta estructura del código que indica una violación de un principio de diseño que puede afectar negativamente en la calidad del código. Algunos ejemplos: clases con demasiados atributos, métodos con demasiados parámetros, una *Cyclomatic complexity* [27] en un método, etc. Hay que tener en cuenta que la existencia de un *code smell* no implica necesariamente que el código no tenga calidad.
- Vulnerabilidades de seguridad: existe un tipo de reglas en este tipo de herramientas especialmente destinadas a determinar vulnerabilidades de seguridad en un código. Por ejemplo, existen analizadores de código que pueden detectar versiones sencillas de vulnerabilidades como el *buffer overflow*, *cross site scripting*, etc. Dentro de estas reglas, también se suelen analizar las versiones de las librerías que se utiliza para mantenerlas

¹ Wikipedia con información básica sobre estas herramientas, dividida por lenguajes de programación: https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

Post en el Blog Software Testing Help, elaborado por el autor del blog: <http://www.softwaretestinghelp.com/tools/top-40-static-code-analysis-tools/>

GitHub, lista adicional mantenida por la comunidad con herramientas relativamente activas y usadas: <https://github.com/mre/awesome-static-analysis>

actualizadas (o al menos no usar la versión de una librería con vulnerabilidades conocidas). Por este motivo, muchas herramientas de análisis de código revisan si las versiones de las librerías usadas por una aplicación tienen alguna vulnerabilidad conocida en la base de datos *Common Vulnerabilities and Exposures* (CVE) [28] o similares. En la siguiente figura podemos ver gráficamente la problemática de las librerías vulnerables (imagen aparecida en el informe sobre vulnerabilidades de BlackDuck [29])



Ilustración 8. Esquema de detección de vulnerabilidades por BlackDuck [29]

- Detección de mal funcionamiento con análisis formal: existen herramientas que van más allá del análisis superficial del código y analizan formalmente el código en busca de bugs. Por ejemplo, estas herramientas más avanzadas suelen usarse para detectar la posibilidad de condiciones de carrera en programas concurrentes.

En ciertas ocasiones no hay una clara distinción entre estos tipos de reglas. Por ejemplo, una regla se podría interpretar como regla de estilo y a la misma vez como regla de buenas prácticas. Muchas herramientas ni siquiera dividen las reglas de esta forma y tienen sus propias categorías. Por ejemplo, SonarQube, divide las reglas en las siguientes secciones: *Bugs*, *Vulnerabilities*, and *Code Smells*.

3.3.2 Niveles de Severidad

Las reglas descritas en el apartado anterior se han clasificado según su importancia, dando como resultado reglas de menor importancia (por ejemplo, las de formato), así como reglas de mayor importancia (por ejemplo, las de seguridad). Cada herramienta tiene una clasificación que ha sido determinada de manera arbitraria según sus desarrolladores. En el caso de SonarQube, las reglas que han sido violadas las denomina *issues* [30], las cuales tienen las siguientes importancias (*severity*) [31]:

- Blocker: indica un bug con una alta probabilidad de impacto en el comportamiento de la aplicación en producción (fuga de memoria, conexiones a la base de datos sin cerrar, etc.). Se debe corregir y arreglar inmediatamente este código.

- **Critical:** indica un bug con una baja probabilidad de impacto a la aplicación en producción, o un problema que representa un potencial problema de seguridad (bloque *catch* vacío, posibilidad de inyección SQL, etc.). Se debe corregir y arreglar inmediatamente este código.
- **Major:** fallo de calidad que puede tener un impacto en la productividad del desarrollador (pieza de código sin cubrir por tests, bloques de código duplicados, parámetros sin usar, etc.).
- **Minor:** fallo de calidad que pueden tener un impacto en la productividad del desarrollador (líneas de código demasiado largas, sentencias *switch* con menos de 3 opciones, etc.).
- **Info:** ni un bug ni un problema de calidad, simplemente una regla que ha sido encontrada.

Conforme se va aumentando el nivel de abstracción de las reglas, éstas podrían ser ambiguas para los diferentes equipos implicados en el desarrollo del producto software. Por este motivo, la mayor parte de las herramientas permiten configuraciones relacionadas con los diferentes aspectos de las reglas, ya que la violación de una regla no implica obligatoriamente que el código deba “arreglarse”:

- **Configuración de las reglas:** en funciones de las necesidades del proyecto, ciertas reglas son configurables
 - **Reglas de formato/estilo del código:** basadas en la longitud máxima de una línea, posición de las llaves de apertura/cierre de bloque (por ejemplo, en la misma línea o siguiente que el *if* o *for*), etc. Por este motivo, existen diferentes *reglas de estilo* que el usuario puede elegir y aplicar según satisfaga sus necesidades.
 - **Reglas de detección de Code Smell:** basadas en umbrales (número máximo de parámetros en un método, número máximo de niveles en una jerarquía de herencia, etc.). Estas métricas suelen ser configurables para que se considere un aviso o un error.
- **Desactivación de ciertas reglas:** los diferentes equipos involucrados pueden llegar al acuerdo de que ciertas reglas no aplican para el proyecto en desarrollo. Por ello, la mayoría de las herramientas ofrecen la posibilidad de desactivar ciertas reglas.
- **El nivel de Severidad de cada regla:** los diferentes equipos involucrados pueden llegar a considerar aceptable la violación de ciertas reglas, clasificando éstas como *warnings* (avisos), mientras que para otros equipos cualquier violación de una regla requiere de una solución inmediata (*error*).
- **Añadir reglas propias:** los diferentes equipos involucrados pueden llegar a decidir qué arquitectura y patrones de diseño aplicar, por ello existen herramientas las cuales per-

miten añadir reglas o patrones propios, para así asegurarse de que dichos patrones elegidos cumplen dentro del proyecto.

3.3.3 Modos de ejecución

Las herramientas de SCA pueden ser utilizadas en diferentes momentos del ciclo de vida de desarrollo software. Por tanto, en función de cuando sean usadas durante el ciclo de vida, éstas ofrecerán unas funcionalidades u otras.

En general, se considera como buena y más recomendable la práctica de usar las herramientas de SCA integradas en el propio entorno de desarrollo, ya que esto reduce considerablemente el tiempo empleado en solventar los problemas generados. El motivo es sencillo: el desarrollador está inmerso completamente en el proyecto teniendo éste en la cabeza, lo que provoca un rápido entendimiento por su parte del inconveniente para resolverlo. Por ello, puede analizar mucho mejor el problema reportado, y ofrecer una solución en un tiempo más inmediato. Las siguientes definiciones hacen referencia a los diferentes momentos en los que se puede utilizar las herramientas SCA:

- Como plugin durante el tiempo de desarrollo: en la actualidad los *Integrated Development Environments* (Entornos de Desarrollo Integrado, IDE) y los editores de texto han evolucionado considerablemente, para proporcionar una mayor satisfacción al usuario. Algunos de los más utilizados son Eclipse [32], Visual Studio Code [33] o Sublime [34]. Estos IDEs o editores, disponen de *plugins*, los cuales se integran con la finalidad de ejecutar las validaciones de SCA en tiempo de desarrollo del código. Como suele ser habitual en los IDEs, todos los avisos y errores tienen enlazados una descripción de por qué lo son. Estos *plugins* no iban a ser menos, también tienen enlaces a descripciones del problema. Algunos de ellos ofrecen la posibilidad de resolver el problema de forma automática sugiriendo diferentes alternativas.
- Como una herramienta de línea de comandos: algunas herramientas SCA se pueden ejecutar en la línea de comandos para realizar un análisis de un proyecto completo. El resultado de estas herramientas se suele mostrar como salida estándar indicando cuales son, donde y en qué ficheros están las violaciones, al igual que lo suelen hacer los compiladores. Estas herramientas permiten la configuración de generar un ficho estructurado (XML o JSON), puesto que existen herramientas que pueden tratar este tipo de ficheros para la realización de diferentes tipos de informes.
- Como plugin de herramientas de Gestión del Ciclo de Vida en proyectos: en general, en los diferentes lenguajes de programación disponen de diferentes herramientas para controlar el ciclo de vida, como los siguientes ejemplos
 - En los proyectos de C/C++ se suele utilizar *make* o *Cmake*.
 - En proyectos Java se suele utilizar Maven o Gradle.

Salvando las distancias, cada una de estas herramientas y todas las que se quedan sin nombrar nos permiten gestionar las diferentes etapas de del desarrollo software de un proyecto: compilación, verificación y construcción. Por estas razones, la mayoría de las

herramientas SCA permiten integrarse dentro de herramientas de control de ciclo de vida, lo que permite que SCA pueda producirse durante las etapas de verificación, en tiempo de ejecución de los tests.

- Como plugin del sistema de integración continua: con el fin de simplificar la consulta de los reportes durante las etapas de validación del sistema de CI, diversas herramientas disponen de *plugins* para los entornos de CI más populares como Jenkins.

3.4 STATIC CODE ANALYSIS EN CIBERSEGURIDAD: SAST

El SCA es una técnica muy utilizada en el ámbito de la ciberseguridad, en realidad, se le ha dado un nombre propio en este ámbito: *Static Application Security Testing* (SAST, en español sería Prueba de Seguridad de Aplicaciones Estáticas). A estas pruebas también se les ha denominado “pruebas de caja blanca”.

OWASP (es un proyecto del que hablaremos detalladamente más adelante) tiene su propia página destinada a este tipo de herramientas [35]. En ella, se mencionan algunas de las ventajas de estas herramientas de SCA para encontrar vulnerabilidades de seguridad:

- Técnica escalable: es una práctica que permite el análisis de forma desatendida para grandes cantidades de código. Una revisión manual equivalente por parte del equipo de desarrollo llevaría mucho tiempo.
- Útil para vulnerabilidades básicas: es muy provechoso para encontrar las vulnerabilidades que las herramientas son capaces de encontrar de forma automática con mucha precisión, como, por ejemplo, el desbordamiento de buffer, la inyección SQL, etc.
- Informes útiles para desarrolladores: proporciona la situación exacta del problema, la justificación de este, además en la mayoría de las ocasiones ofrece posibilidades para solucionarlo pudiéndose aplicar la misma de manera automática.

No obstante, no es una buena práctica dejar de pensar en las limitaciones asociadas a este tipo de herramientas, ya siempre se debe comprender que la seguridad completa no existe:

- Vulnerabilidades no detectadas: existen multitud de vulnerabilidades que son muy difíciles de encontrar analizando el código de manera automática. Por ejemplo, las relacionadas con la autenticación, con el control de acceso, con el uso inseguro de herramientas criptográficas, etc. Según los diversos estudios relacionados con SCA, las herramientas que realizan este tipo de análisis encuentran un porcentaje de los problemas de seguridad menor de lo deseado debido a la complejidad de la automatización, aunque están mejorando constantemente.
- Falsos positivos: en todos los ámbitos las herramientas pueden tener un número elevado de falsos positivos. Las herramientas de SCA no se quedan atrás teniendo un resultado más elevado de lo deseado, lo que provoca que el desarrollador invierta un tiempo en analizar estos resultados. Esto nos lleva a la conclusión de que el análisis manual es fundamental, ya que la automatización va mejorando poco a poco de la mano humana.

3.4.1 SonarQube



Una de las suites de herramientas más completas para analizar el código fuente de un programa es el conjunto de herramientas de SonarSource [36], del cual se ha seleccionado para este Trabajo la herramienta SonarQube [37].

En los puntos **Reglas del SCA y SAST, Niveles de Severidad, y Modos de ejecución**, se han ido explicando las características y funcionalidades generales que describen y definen a esta herramienta. A continuación, se resaltan las principales características de esta herramienta son:

- Múltiples lenguajes: soporta de múltiples lenguajes de programación. Java, PHP, C#, TypeScript, JavaScript, C/C++ y muchos más.
- Inspección continua: una de sus principales características es que favorece la inspección continua del código ya que ofrece dashboards y gráficas de evolución. Además, como se ha indicado previamente, diferencia entre los *issues* presentes en el código y los introducidos (o reducidos) en el último cambio. Estas funcionalidades favorecen el uso de la herramienta dentro de un enfoque de gestión de deuda técnica y mejora continua.
- Quality Gates: también dispone del concepto de puerta de calidad (quality gate), a modo de semáforo que puede usarse en sistemas de continuous delivery para controlar la generación de una nueva versión o el paso o el despliegue automático.
- Sistema de plugins: existen gran cantidad de plugins gratuitos y comerciales. Los desarrolladores también pueden incluir sus propios plugins.
- Versión libre y empresarial: SonarQube puede usarse gratuitamente en su versión software libre. Además, dispone de una versión empresarial con funcionalidades avanzadas.

Para este Trabajo, se ha implementado esta herramienta, tanto como un servidor independiente desplegado en Docker [38], como en su forma de plugin [39], dentro del servidor de Jenkins para su integración entre el servidor dedicado SonarQube y el servidor CI/CD Jenkins.

3.4.2 OWASP Dependency Check



Es una herramienta que tiene como objetivo controlar y minimizar las vulnerabilidades provenientes de los frameworks o librerías de terceros. Su finalidad es identificar las dependencias de los proyectos y comprobar si existe alguna vulnerabilidad conocida. Actualmente, Dependency-Check da soporte a los lenguajes Java y .NET, junto con un soporte experimental a Ruby, Node.js y Python, y un soporte limitado a C/C++ [40].

Esta herramienta funciona recopilando información generada por los analizadores que utiliza, clasificándola como Evidencias de tres tipos: productor, producto, y versión. En el caso de Java, utiliza un analizador llamado *JarAnalyzer* que recopila información del *Manifest*, *pom.xml*, y los nombres de los paquetes dentro de los archivos JAR escaneados [41].

Para este Trabajo se ha implementado esta herramienta en su forma de plugin dentro del servidor de Jenkins [42].

3.4.3 Clair: Static Analysis of Vulnerabilities

Clair es un proyecto *open-source* que actúa como un analizador de seguridad de contenedores e imágenes Docker. Ha sido desarrollado por CoreOS [43].



Es una herramienta poderosa y extensible que inspecciona las imágenes de los contenedores en busca de vulnerabilidades de seguridad conocidas y permite a los desarrolladores crear servicios que escanean los contenedores en busca de amenazas y vulnerabilidades de seguridad.

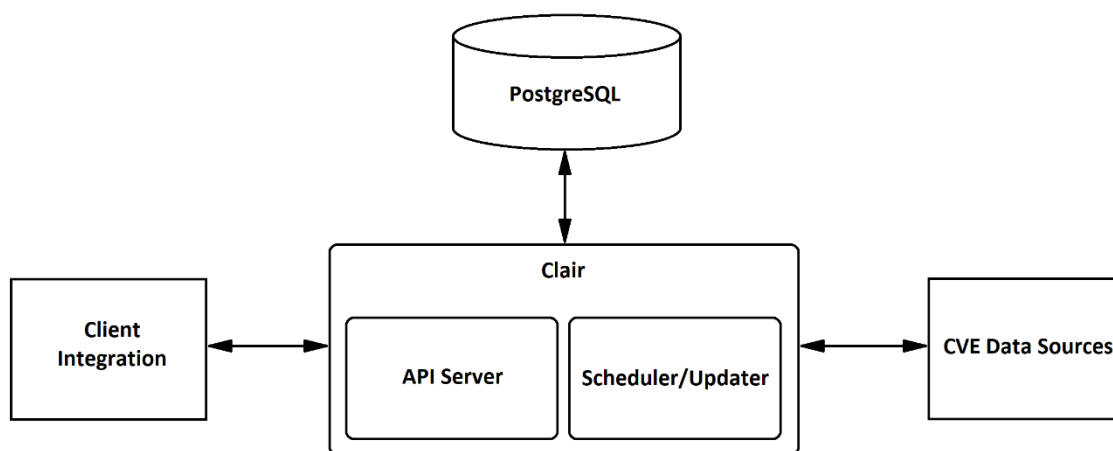


Ilustración 9 Esquema del framework de Clair

El servicio de Clair consiste en tres componentes principales:

1. **Servidor API:** Clair no tiene una interfaz de línea de comandos o una interfaz de usuario con la que poder interactuar directamente, por eso provee una interfaz API REST para interactuar con Clair a través de los clientes de integración compatibles, y así poder realizar el escaneo.
2. **Base de datos PostgreSQL:** utilizada para almacenar toda la información relevante a las vulnerabilidades CVE para la realización de los escaneos.
3. **Fuente de información de vulnerabilidades CVE:** hay muchas fuentes *Common Vulnerability and Exposure* (CVE) con las que trabaja Clair, entre otras están *Debian Security Bug Tracker*, *Ubuntu CVE Tracker*, y *NIST NVD*.

Como componente interno en Clair se encuentra un actualizador de las fuentes CVE que está sincronizado con dichas fuentes y se encarga de actualizar la base de datos PostgreSQL con las vulnerabilidades conocidas.

Clair funciona realizando un escaneo estático, esto quiere decir que escanea la imagen y no la instancia del contenedor en ejecución. Las imágenes Docker constan de una o más capas, que también son imágenes y se almacenan en el registro de Docker como blobs. Esto hace que Clair analice estas capas o una colección de múltiples capas (una imagen).

Como ya se ha comentado, Clair solo proporciona una interfaz API REST, por lo que existen herramientas de integración como clientes para interactuar con Clair [44]. La mayoría de ellas nos ayudan a integrar el escaneo estático de contenedores e imágenes Docker en nuestros sistemas de CI/CD. En los siguientes puntos se describen las dos herramientas seleccionadas para que han encajado mejor para la integración en el Pipeline DevSecOps.

Para este Trabajo, se ha implementado esta herramienta como un servidor independiente desplegado en Docker [45] junto con la base de datos PostgreSQL recomendada en su repositorio de GitHub [46].

3.4.3.1 Clairctl

Es una herramienta de línea de comandos ligera que funciona entre el servidor API REST de Clair y los registros de imágenes Docker, como pueden ser Docker Hub, Docker Registry, o Quay.io. Una de sus características es que funciona como un proxy inverso para la autenticación, además de proporcionar un comando para comprobar la salud y poder determinar el correcto funcionamiento del servicio: `clairctl health`. También es capaz de generar informes HTML con los resultados del análisis [47].

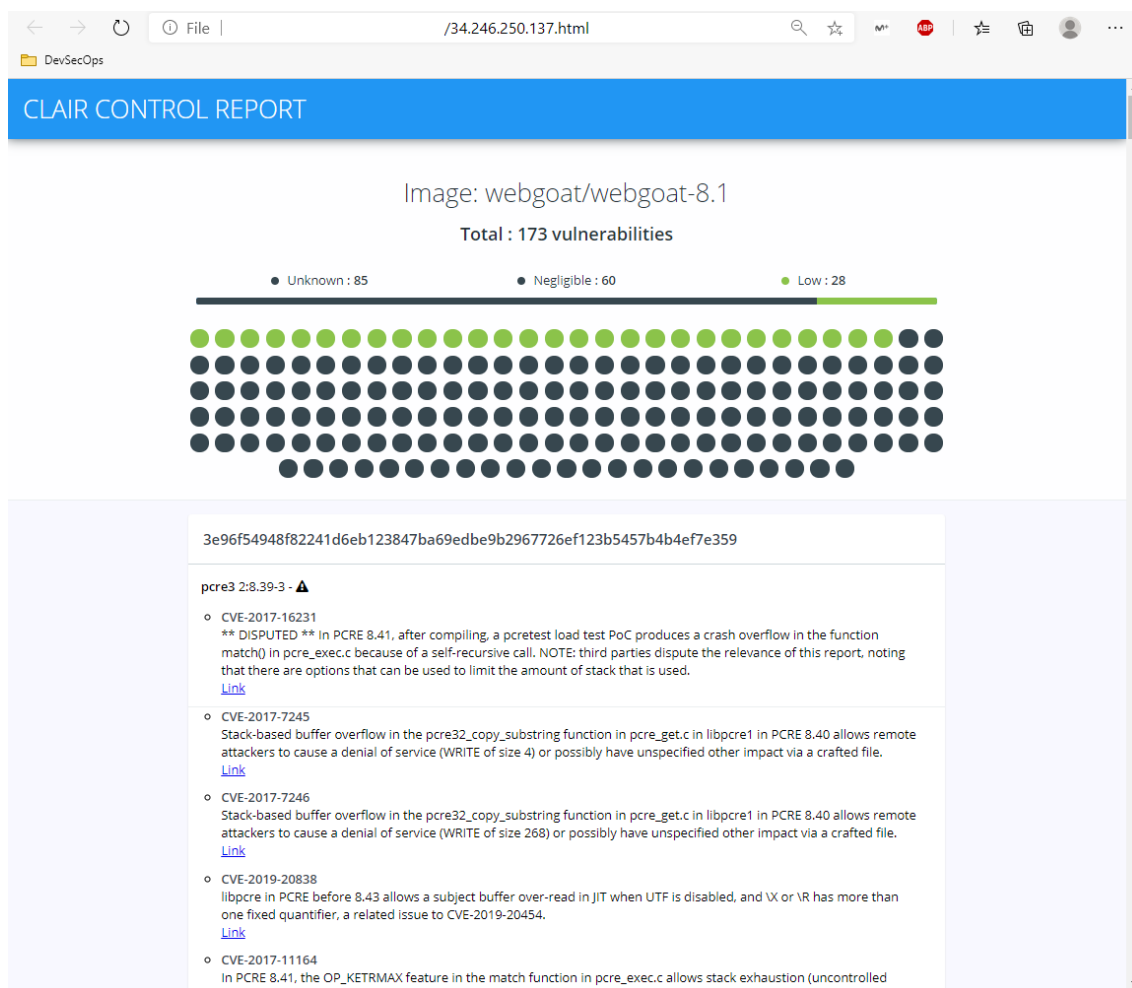


Ilustración 10 Reporte HTML del análisis Clair generado por Clairctl

Para este Trabajo, se ha optado por implementar su imagen Docker para poder desplegarla en el ecosistema DevSecOps y tener un contenedor exclusivo para su funcionamiento [47].

3.4.3.2 Clair-Scanner

Es una herramienta de integración con Clair para la realización de escaneos de imágenes y contenedores Docker de manera local. Clair no tiene una herramienta simple que realice los escaneos localmente de una manera simple o sencilla, por ello Clair-Scanner se encarga de interactuar con Clair para llevarlo a cabo y realizar una comparación de las vulnerabilidades con una lista blanca (whitelist) [48].

```
ne > #64
+ chmod +x clair-scanner
+ ./clair-scanner --ip=172.18.0.4 --clair=http://clair:6060 webgoat/webgoat-8.1
2020/08/29 20:46:15 [0:32m[INFO] [0m Start clair-scanner[0m
2020/08/29 20:46:18 [0:32m[INFO] [0m Server listening on port 9279[0m
2020/08/29 20:46:18 [0:32m[INFO] [0m Analyzing 08bf86d6624450c487db18071224c88003d970848fb8c5b2b07df27e3f6869b2[0m
2020/08/29 20:46:18 [0:32m[INFO] [0m Analyzing 6344b42c1c8e3c1134fa2dfe0ae114199376a63f38bc41640d426cca90bcb26d[0m
2020/08/29 20:46:18 [0:32m[INFO] [0m Analyzing 55554b21af5ae92da6b70a0767ea4e58ba9e5a14e159858f107103218966000c[0m
2020/08/29 20:46:18 [0:32m[INFO] [0m Analyzing 6aca162c2dcdf2c2dc6a3983b4584e9b09802ff8f6530e733f828438c3827ae9[0m
2020/08/29 20:46:18 [0:32m[INFO] [0m Analyzing a6175e7e36da80235b11eb57b283c18d48db28e9e9656354fe9e18e42263a67f[0m
2020/08/29 20:46:18 [0:32m[INFO] [0m Analyzing fc5cbdb61ba1bdefd682f3821c9424935483e3013e1581d02d975b752a19aea[0m
2020/08/29 20:46:18 [0:32m[INFO] [0m Analyzing affa2b3d6ad019d130f99b310633f6b08fb46c1e2d71ac45e1f41b2a920a10ed[0m
2020/08/29 20:46:18 [0:32m[INFO] [0m Analyzing 21e9982ef9d15e421de49108be3ba075f6c300a991c9b7633113c7b1b5585c21[0m
2020/08/29 20:46:18 [0:32m[INFO] [0m Analyzing 946f60fd1599836806c622330d66e1f7ea834da507d1d1d4e661d4830906742[0m
2020/08/29 20:46:18 [0:33m[WARN] [0m Image [webgoat/webgoat-8.1] contains 173 total vulnerabilities[0m
2020/08/29 20:46:18 [0:31m[ERRO] [0m Image [webgoat/webgoat-8.1] contains 173 unapproved vulnerabilities[0m
```

STATUS	CVE SEVERITY	PACKAGE NAME	PACKAGE VERSION	CVE DESCRIPTION	
[1;31mUnapproved[0m	[0m Low	CVE-2018-11236	glibc	2.24-11+deb9u3	stdlib/canonicalize.c in the GNU C Library (aka glibc or libc6) 2.27 and earlier, when processing very long pathname arguments to the realpath function, could encounter an integer overflow on 32-bit architectures, leading to a stack-based buffer overflow and, potentially, arbitrary code execution. https://security-tracker.debian.org/tracker/CVE-2018-11236
[1;31mUnapproved[0m	[0m Low	CVE-2018-7169	shadow	1:4.4-4.1	An issue was discovered in shadow 4.5. newgidmap (in shadow-utils) is setuid and allows an unprivileged user to be placed in a user namespace where setgroups(2) is permitted. This allows an attacker to remove themselves from a supplementary group, which may allow access to certain filesystem paths if the administrator has used "group blacklisting" (e.g., chmod g-rwx) to restrict access to paths. This flaw effectively reverts a security feature in the kernel (in particular, the /proc/self/setgroups knob) to prevent this sort of privilege escalation. https://security-tracker.debian.org/tracker/CVE-2018-7169

Ilustración 11 Reporte por consola del análisis Clair generado por Clair-Scanner

Como se puede observar en la imagen anterior, la interacción entre Clair y Clair-Scanner es por línea de comandos, por lo que Clair genera el resultado del análisis como salida de texto por línea de comandos, donde se muestra las capas que se han analizado y una tabla detallada de las vulnerabilidades encontradas.

Para este Trabajo, se ha implementado esta herramienta como ejecutable durante la etapa de análisis de vulnerabilidades, haciendo que el servidor Jenkins se descargue la herramienta y la ejecute en el momento preciso [48].

3.5 DYNAMIC APPLICATION SECURITY TESTING (DAST)

Dynamic Application Security Testing (DAST, en español sería Pruebas de Seguridad de Aplicaciones Dinámicas), también denominadas como pruebas de "caja negra", son las pruebas destinadas a la búsqueda de vulnerabilidades de seguridad en aplicaciones en ejecución, normalmente en aplicaciones web, por lo que no tienen acceso al código fuente y, por lo tanto, tienen que buscar y encontrar las vulnerabilidades realizando ataques contra la aplicación cuando ésta se encuentra en su etapa final del ciclo de vida del desarrollo software: despliegue [49].

La manera que tienen de encontrar estas vulnerabilidades es realizando ataques conocidos, como *Cross-Site Scripting*, *SQL Injection*, *Directory Traversal*, configuraciones inseguras, y vulnerabilidades de ejecución remota de comandos. También puede destacar los problemas de tiempo de ejecución que no se pueden identificar mediante análisis estático, por ejemplo, problemas de autenticación y configuración del servidor, así como fallas visibles solo cuando un usuario conocido inicia sesión [50].

Este tipo de análisis se realiza automatizando las herramientas involucradas en el proceso, haciendo que sea una tarea desatendida para los responsables.

3.5.1 OWASP ZAP

Es una herramienta *open-source* que está diseñada específicamente para probar la de seguridad en aplicaciones web, escaneando las vulnerabilidades [51].



En esencia, ZAP es lo que se conoce como un “man-in-the-middle proxy”. Se encuentra entre el navegador web del tester y la aplicación web, dando lugar a la interceptación e inspección de los mensajes enviados entre el navegador y la aplicación web, posibilitando la modificación del contenido si es necesario, y finalmente reenviar esos paquetes al destino.



Se puede utilizar como una aplicación independiente y como un proceso Daemon (demonio en escucha).

Ilustración 12 Configuración servicio OWASP ZAP “man-in-the-middle” [79]

ZAP proporciona funcionalidad para una gran variedad de niveles de habilidad, desde nuevos probadores en pruebas de seguridad hasta especialistas en pruebas de seguridad.

Tiene versiones para cada sistema operativo y Docker, desacoplando la infraestructura software para poder usarlo.

Para este Trabajo, se ha implementado esta herramienta como un servidor independiente desplegado en Docker [52].

3.6 APLICACIÓN WEB VULNERABLE: WEBGOAT

Para poder llevar a cabo este Trabajo, es necesario disponer de un proyecto software que esté desarrollado y que contenga una calidad más bien baja la cual genere tanto errores como vulnerabilidades de seguridad. Por ello, se ha realizado una búsqueda para determinar cuál es la mejor aplicación posible para encajar en el marco de este Trabajo, la cual ha dado como resultado la elección de la aplicación insegura **WebGoat** creada bajo el amparo de OWAPS.

Open Web Application Security Project (OWASP, en español Proyecto abierto de seguridad de aplicaciones web) es un proyecto de código abierto nacido para determinar y remediar las causas que convierten al software en inseguro, dando como resultado la creación y respaldo del proyecto de la Fundación OWASP, siendo esta un organismo sin ánimo de lucro encargado de apoyar y gestionar los proyectos e infraestructura de que nacen de OWASP, además de trabajar en la creación de artículos, metodologías, documentación (Guía OWASP), herramientas (WebGoat) y tecnologías que sean libres y gratuitas para cualquiera [53].

3.6.1 WebGoat

La seguridad de las aplicaciones web es difícil de aprender y practicar, a lo que hay que sumar la indisponibilidad de aplicaciones web completas, como librerías o bancos en línea, que se puedan usar para buscar vulnerabilidades. Por otro lado, los profesionales de la seguridad necesitan frecuentemente probar las herramientas en un entorno que se sabe que es vulnerable para así asegurarse de que funcionen correctamente. Por tanto, es necesario disponer de un entorno legal y seguro para poder llevar a cabo las pruebas necesarias.

WebGoat es una aplicación deliberadamente desarrollada para ser insegura con la que permitir a los desarrolladores, y en general a cualquier persona, probar las vulnerabilidades comúnmente encontradas en las aplicaciones basadas en Java y que hacen uso de componentes *open source*.

Por todo ello, el objetivo principal del proyecto WebGoat es simple: crear un entorno interactivo de enseñanza de la seguridad de las aplicaciones web [54].

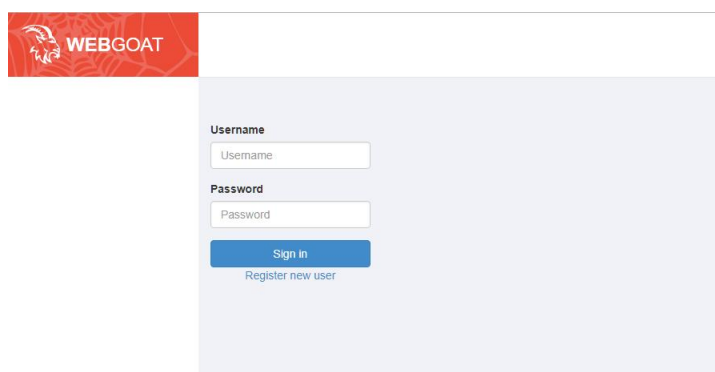


Ilustración 13 Aplicación WebGoat desplegada y funcionando

3.7 KUBERNETES

Es un sistema *open source* para la gestión de las aplicaciones desplegadas en contenedores, normalmente distribuidas en diferentes máquinas, el despliegue de estos, así como sus cargas de trabajo y servicios. Kubernetes, conocido comúnmente en inglés como “K8s”, tiene un ecosistema grande que proporciona mecanismos básicos para el despliegue de las aplicaciones, incluido el despliegue automatizado, el mantenimiento de estas, y su escalado en función de la carga de trabajo [55] [56].

Fue desarrollado originalmente por Google, quien liberó el proyecto en el año 2014 después de década y media de experiencia realizando cargas de trabajo escaladas en entornos de producción. Actualmente el proyecto se encuentra mantenido por *Cloud Native Computing Foundation* (CNCF) [57].

Uno de los objetivos de este Trabajo es la migración del entorno CI/CD DevSecOps hacia un ecosistema desplegado en Kubernetes, para lo cual, se ha decidido hacer uso de la herramienta Kompose.

3.7.1 Kompose

Es una herramienta creada para facilitar a los usuarios familiarizados con *Docker-Compose* la migración de sus ecosistemas al sistema Kubernetes [58].

La herramienta funciona leyendo el esquema YAML de *Docker-Compose* ya creado para hacer interpretación de todo lo configurado y convertir cada elemento en los diferentes ficheros YAML de Kubernetes listos para ser delegados en el sistema. Actualmente la herramienta solo es compatible con los esquemas YAML de versiones 1, 2, y 3 de *Docker-Compose*. Si la versión es más reciente mostrará un error al intentar usarla [59].

Kompose hace uso de otras herramientas para poder ser usada. En su caso, será necesaria la instalación y configuración adecuada tanto de la herramienta kubectl [60] y de minikube [61] [62].

Para este Trabajo, se ha implementado esta herramienta realizando la configuración necesaria. Por ello, ha sido necesario hacer alguna modificación en el esquema YAML de *Docker-Compose* creado para este Trabajo y disponible en el Anexo 1: Docker Compose File script.

4 DESARROLLO DEL PROYECTO

4.1 ANÁLISIS

El objetivo prioritario de este Trabajo es el de integrar la seguridad como un agente más del ciclo de vida del desarrollo software, incluyendo a las operaciones y a los desarrolladores en la concienciación de la seguridad, haciendo de esta manera que todas estas partes sean partícipes de forma activa y colaborativa, resaltando así que la seguridad no es algo bloqueante, ni produce retrasos en las entregas, sino una mejora preventiva, puesto que previene problemas posteriores y minimiza mantenimientos, lo cual incide en el ahorro de costes y en la percepción de la calidad por parte del usuario final.

Dado que no es motivo de este proyecto el desarrollo de una aplicación web desarrollada en Docker y/o Kubernetes, se ha realizado una búsqueda para seleccionar dicha aplicación. Como ya se ha comentado anteriormente, se ha decidido utilizar la aplicación WebGoat por ser una aplicación específicamente diseñada para ser insegura.

También se ha hecho uso de una imagen EC2 de Ubuntu en AWS, en el cual se ha utilizado Docker para el despliegue del servidor de Integración Continua, así como de las herramientas de seguridad descritas y empleadas para este Trabajo.

4.2 DISEÑO DEL CICLO DE VIDA DEL SOFTWARE DEVSECOPS

Siendo el objetivo principal la creación de un ciclo de vida DevSecOps como Pipeline en un entorno CI/CD, el desarrollo de este Trabajo se apoya en la cultura DevOps para lograr dicho objetivo, teniendo como referencia las etapas en las que se basa la metodología DevOps para su ejecución. en base a ellas, se ha creado un ciclo de vida del desarrollo software seguro (SSDLC [63]) DevSecOps. A continuación, se exponen las etapas DevOps, en las que se basa este SSDLC DevSecOps, además de las que lo componen.



Ilustración 14 Etapas generales en DevOps

Ilustración 15 Etapas DevSecOps en este Trabajo

En el diseño del ciclo de vida DevSecOps se pueden ver las etapas implementadas de manera automatizada, y su asociación con las etapas DevOps de las que descienden.

4.2.1 Etapas DevOps

Las etapas de un ciclo de vida DevOps son las siguientes [64]:

4.2.1.1 Planificación

Es una etapa en la que se intentan establecer mecanismos para que los participantes, directos o indirectos, puedan aportar de forma continua sus ideas para añadir valor en cada interacción. Se establecen los criterios de aceptación, funcionalidades, los requisitos y políticas (incluyendo referentes a seguridad), posibles modelos de amenazas, etc.

4.2.1.2 Desarrollo

Esta etapa es la de materialización del código fuente del proyecto software, como pueden ser monitorización del código mediante un control del repositorio con control de versiones de este, diseño de infraestructura, automatización de procesos, diseño y automatización de las pruebas. En esta etapa, tal y como se ha comentado en el estado del arte, se plantea la codificación de pruebas unitarias automatizadas, además de la utilización de IDEs en los que se puedan implementar *plugins* para la realización de SAST durante el proceso de desarrollo para minimizar las vulnerabilidades de seguridad durante esta etapa.

4.2.1.3 Integración Continua (CI)

Esta etapa es una de las más críticas en los ciclos DevOps, ya que en ella se implementan los mecanismos automatizados de revisión, validación, y pruebas.

4.2.1.4 Release

En esta etapa se realiza la generación de los binarios del proyecto software donde se incluye la integración de la nueva versión del código fuente de las etapas anteriores.

4.2.1.5 Despliegue Continuo (CD)

En esta etapa se realiza el despliegue automatizado de la aplicación de tal manera que todo el proceso se resuelva de la manera más conveniente para el proyecto software.

4.2.1.6 Operaciones

En esta etapa, se aplican los controles para garantizar que el Sistema funciona correctamente según las especificaciones, en rendimiento, utilización, funcionalidad, experiencia de usuario, etc. Además, se realiza el mantenimiento del Sistema que aloja la nueva release.

4.2.1.7 Feedback Continuo

En esta etapa es crucial la comunicación entre todos los integrantes del proyecto software, así como el análisis y el tratamiento de la información generada por todos los sistemas y herramientas empleados durante todo el ciclo de vida del proyecto software. Es muy importante abordar esta etapa con aquellas herramientas que satisfagan las necesidades del proyecto para así poder tener la información necesaria para la mejor alimentación del ecosistema.

4.2.2 Etapas DevSecOps

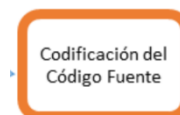
Las etapas de un ciclo de vida DevSecOps diseñadas son las siguientes:

4.2.2.0 Planificación

Esta etapa no aparece en el diagrama del apartado 4.2 ya que se considera como etapa inicial del ciclo de vida del desarrollo software DevSecOps, habiendo sido ya realizada puesto que no es motivo de este Trabajo la realización de esta.

4.2.2.1 Codificación del Código fuente

Se realiza la codificación del código fuente resultante de la etapa general de Planificación DevOps, la cual se ha realizado mínimamente para solventar un error del código fuente existente en el proyecto en GitHub.



4.2.2.2 Subida al Repositorio

Se plantea una subida al repositorio para el control de versiones y la integración continua del código en el repositorio principal de desarrollo. Posteriormente realizar un análisis de las dependencias y un análisis SAST.



4.2.2.3 Monitorización de Repositorios

Se monitoriza el repositorio principal para integrar los cambios en este, para que cuando haya un cambio el servidor CI Jenkins haga una copia del repositorio para la realización del análisis SAST el cual tendrá un resultado de aceptación o rechazo de mínima calidad. En esta etapa se aplican las políticas de seguridad definidas en la etapa de Planificación.



4.2.2.4 Análisis de Seguridad de las Dependencias

Se realiza un análisis de la seguridad de las dependencias utilizadas en el ciclo de vida del proyecto software, generando un informe del resultado del análisis. Se ha utilizado e integrado la herramienta OWASP Dependency-Check de manera automatizada en el Pipeline.



4.2.2.5 SAST

Se realiza un análisis SAST completo del código a integrar el cual tendrá un resultado de aceptación o rechazo de mínima calidad, debido a que se aplican las políticas de seguridad definidas en la etapa de Planificación. Se ha utilizado e integrando la herramienta SonarQube Scanner (plugin de SonarQube para Jenkins) de manera automatizada en el Pipeline.



4.2.2.6 Creación de Binarios e Imágenes Docker

Se realiza la creación automatizada de los binarios haciendo uso de Maven [18], para posteriormente realizar la creación de la imagen Docker a partir de los mismos haciendo uso del fichero Dockerfile, donde se define la construcción de la imagen que se va a desplegar posteriormente.



4.2.2.7 Análisis de la Seguridad en Imágenes Docker

Se realiza una etapa muy parecida a la SAST, pero orientada al análisis de vulnerabilidades de imágenes Docker. Se ha utilizado e integrando la herramienta Clair de manera automatizada en el Pipeline.



4.2.2.8 Despliegue de la imagen Docker

Se ha realizado un despliegue local de la aplicación después de que se haya generado una versión de la imagen Docker. Se ha utilizado la herramienta Docker para la realización del despliegue de la aplicación de manera local.



4.2.2.9 DAST

Se ha realizado una etapa adicional en la que se realizan análisis dinámicos DAST automatizados para evaluar los requisitos de seguridad establecidos en la etapa de Planificación. Se ha utilizado e integrado la herramienta OWASP ZAP de manera automatizada en el Pipeline



4.2.2.10 Feedback Continuo

Se ha prescindido de este análisis en profundidad ya que solo se ha tomado la información reportada por Jenkins en cada una de las ejecuciones o *Builds* del *Job* implementado como Pipeline buscando la funcionalidad DevSecOps. En este caso, haciendo únicamente análisis y tratamiento de la información proporcionada por Jenkins, se ha conseguido integrar satisfactoriamente todas las herramientas expuesta en este Trabajo, solucionando todas las dificultades encontradas en forma de errores o incompatibilidades.



4.3 IMPLANTACIÓN

Para el desarrollo de este proyecto, se ha configurado un entorno de trabajo, el cual es una instancia EC2 de AWS con una AMI base de Ubuntu 18.04 LTS.

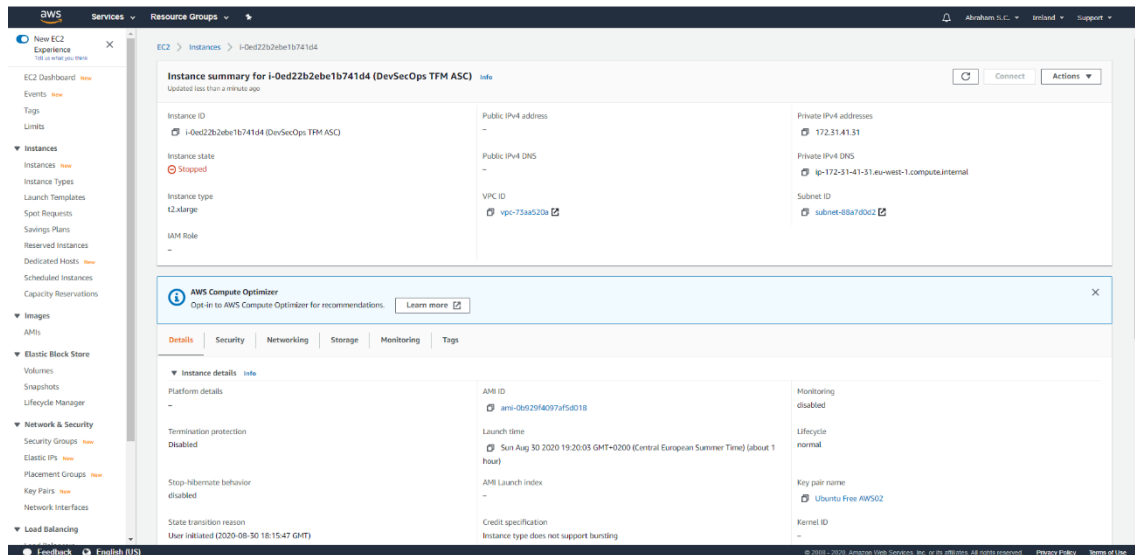


Ilustración 16. Configuración Instancia Ubuntu usada en AWS

Como se puede observar, se ha configurado la instancia de tipo t2.xlarge, proporcionando 4 CPUs virtuales de procesamiento y 16GiB de memoria RAM. Se ha seleccionado un almacenamiento del tipo SSD de 30 GiB. En cuanto a la configuración de seguridad, la que se refiere a las conexiones, se ha dejado abierto el puerto 22 del protocolo TCP para conexiones SSH (utilizado para conectarse a la máquina remotamente mediante Putty [65], que es un cliente SSH) pero para poder conectarse a la máquina es necesario utilizar una pareja de claves generadas por el propio AWS.

Cuando la máquina ya es funcional, se ha decidido utilizar *Docker-Compose* para el despliegue tanto de todo el ecosistema que compone el ciclo de vida DevSecOps del que nace el Pipeline, mediante el fichero de configuración de despliegue *docker-compose.yml*, cuyo contenido del fichero se encuentra en el Anexo 1.

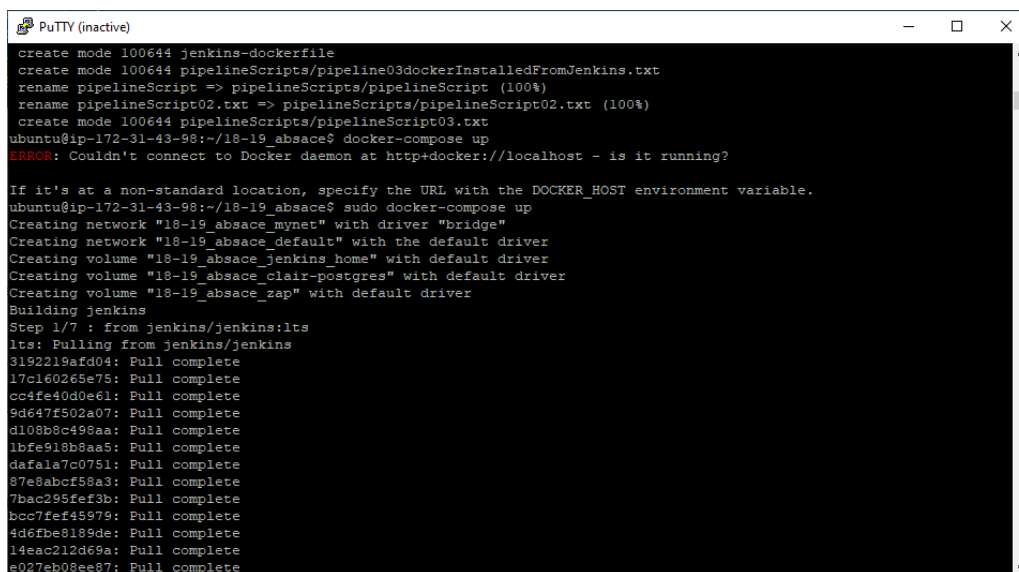
4.3.1 Docker-Compose: orquestación del entorno DevSecOps

Este Trabajo ha generado una serie de archivos asociados a las tecnologías usadas, como es el caso de un YAML para la orquestación de Docker-Compose, un Dockerfile para la creación de la imagen Docker de Jenkins con la instalación de Docker, archivos de configuración para Clair, etc. Todos los archivos se han gestionado en el repositorio creado para cada TFM [66] creado por los responsables del MCyP de la URJC, teniendo cada alumno su repositorio.

La puesta en marcha del entorno DevSecOps se ha realizado haciendo uso de las herramientas dockerizadas, por tanto, se ha utilizado Docker-Compose para su orquestación, describiendo y configurando todos los servicios necesarios en un fichero YAML, el cual se utiliza para la configuración inicial y sucesivas puestas en marcha del entorno. Para ello, haciendo uso del protocolo

SSH para conectarse a la máquina de AWS donde se ha hecho una clonación del repositorio, se utiliza el comando `docker-compose up` se empiezan a crear los contenedores a partir de las imágenes indicadas en el YAML. En el caso de que las imágenes no estén en el registro local de Docker, se descarga las necesarias para poder crear los contenedores, tal y como puede verse en las imágenes que se encuentran más adelante.

Una vez se han descargados todas, crea los contenedores y comienzan a funcionar las aplicaciones (Jenkins, SonarQube, PostgreSQL, Clair, Clairctl, OWASP ZAP) como si de nativas se trataran, pero con la ventaja de estar aisladas.



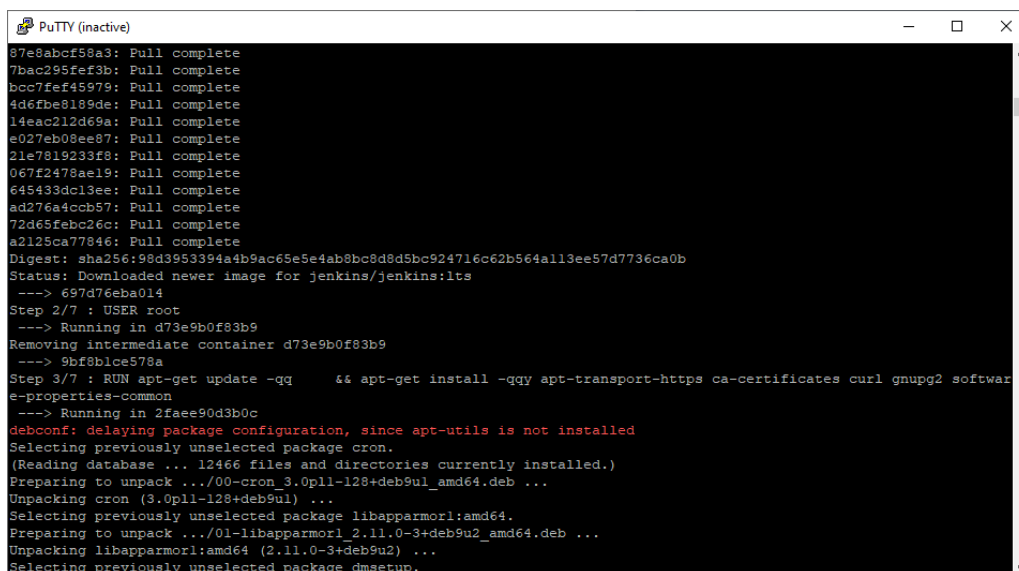
```

create mode 100644 jenkins-dockerfile
create mode 100644 pipelineScripts/pipeline03dockerInstalledFromJenkins.txt
rename pipelineScript => pipelineScripts/pipelineScript (100%)
rename pipelineScript02.txt => pipelineScripts/pipelineScript02.txt (100%)
create mode 100644 pipelineScripts/pipelineScript03.txt
ubuntu@ip-172-31-43-98:~/18-19_absace$ docker-compose up
ERROR: Couldn't connect to Docker daemon at http://localhost:2376: is it running?

If it's at a non-standard location, specify the URL with the DOCKER_HOST environment variable.
ubuntu@ip-172-31-43-98:~/18-19_absace$ sudo docker-compose up
Creating network "18-19_absace_mynet" with driver "bridge"
Creating network "18-19_absace_default" with the default driver
Creating volume "18-19_absace_jenkins_home" with default driver
Creating volume "18-19_absace_clair-postgres" with default driver
Creating volume "18-19_absace_zap" with default driver
Building jenkins
Step 1/7 : from jenkins/jenkins:lts
lts: Pulling from jenkins/jenkins
3192219afd04: Pull complete
17c160265e75: Pull complete
cc4fe40d0e61: Pull complete
9d647f502a07: Pull complete
d108b8c498aa: Pull complete
1bfe918b8aa5: Pull complete
dafala7c0751: Pull complete
87e8abcf58a3: Pull complete
7bac295fef3b: Pull complete
bcc7fef45979: Pull complete
4d6f8e8189de: Pull complete
14eac212d69a: Pull complete
e027eb08ee87: Pull complete

```

Ilustración 17 Despliegue ecosistema DevSecOps usando Docker-Compose vía Putty (1)



```

87e8abcf58a3: Pull complete
7bac295fef3b: Pull complete
bcc7fef45979: Pull complete
4d6f8e8189de: Pull complete
14eac212d69a: Pull complete
e027eb08ee87: Pull complete
21e7819233f8: Pull complete
067f2478ae19: Pull complete
645433dc13ee: Pull complete
ad276a4ccb57: Pull complete
72d65febc26c: Pull complete
a2125ca77846: Pull complete
Digest: sha256:98d3953394a4b9ac65e5e4ab8bc8d8d5bc924716c62b564a113ee57d7736ca0b
Status: Downloaded newer image for jenkins/jenkins:lts
--> 697d76eba014
Step 2/7 : USER root
--> Running in d73e9b0f83b9
Removing intermediate container d73e9b0f83b9
--> 9bf9b1ce578a
Step 3/7 : RUN apt-get update -qq && apt-get install -qqy apt-transport-https ca-certificates curl gnupg2 software-properties-common
--> Running in 2faee90d3b0c
debconf: delaying package configuration, since apt-utils is not installed
Selecting previously unselected package cron.
(Reading database ... 12466 files and directories currently installed.)
Preparing to unpack .../00-cron 3.0p11-128+deb9u1_amd64.deb ...
Unpacking cron (3.0p11-128+deb9u1) ...
Selecting previously unselected package libapparmor1:amd64.
Preparing to unpack .../01-libapparmor1 2.11.0-3+deb9u2_amd64.deb ...
Unpacking libapparmor1:amd64 (2.11.0-3+deb9u2) ...
Selecting previously unselected package dmsetup.

```

Ilustración 18 Despliegue ecosistema DevSecOps usando Docker-Compose vía Putty (2)

```

PuTTY (inactive)
Removing intermediate container 6919b532b0ce
--> 46bf8f46790b

Successfully built 46bf8f46790b
Successfully tagged l8-19 absace_jenkins:latest
WARNING: Image for service jenkins was built because it did not already exist. To rebuild this image you must use `docker-compose build` or `docker-compose up --build`.
Pulling sonarqube (sonarqube:lts)...
lts: Pulling from library/sonarqube
bf5952930446: Pull complete
092c9b8e633f: Pull complete
0b793152b850: Pull complete
b612fb485c1a: Pull complete
1b871e2c81aa: Pull complete
a2210bf5dff8: Pull complete
88222cd31429: Pull complete
923c504afa07: Pull complete
da467387d735: Pull complete
Digest: sha256:e2def97ce0c42fd665bc0ea055cd3e43abef7d63be1c119931020461c22bfe13
Status: Downloaded newer image for sonarqube:lts
Pulling postgres (postgres:9.6)...
9.6: Pulling from library/postgres
75cb2ebf3b3c: Pull complete
3ca6415d2bca: Pull complete
ac08e6372a7b: Pull complete
b4394f9ce95ce: Pull complete
6edcd5da08e3: Pull complete
3380dcb7db08: Pull complete
c7c147d9c90d: Pull complete
08ae47fef758: Pull complete
988b755ec587: Pull complete

```

Ilustración 19 Despliegue ecosistema DevSecOps usando Docker-Compose vía Putty (3)

```

PuTTY (inactive)
Digest: sha256:9aa0b86ae3be8decf922441b913e8914e840c652b6880a642f42f98f5e2aaeaf
Status: Downloaded newer image for postgres:9.6
Pulling clair (quay.io/coreos/clair:v2.0.6)...
v2.0.6: Pulling from coreos/clair
4fe2ade4980c: Pull complete
2e793f0e8e8a: Pull complete
77995fba1918: Pull complete
4495499e856d: Pull complete
0ff8f8e34aa6: Pull complete
329372afb8d2: Pull complete
db7b9f552f3f: Pull complete
37f662d76955: Pull complete
Digest: sha256:931edc5abc036bcc6cd8316e54217958fe27c00f71bbf45c72814563c6acd088
Status: Downloaded newer image for quay.io/coreos/clair:v2.0.6
Pulling clairctl (jsgsquare/clairctl:latest)...
latest: Pulling from jsgsquare/clairctl
6f821164d5b7: Pull complete
9c0celf075fd: Pull complete
Digest: sha256:d07bd8298892355f9b7c84861e5d49250ed2e67881151c75006e600bfc04b4b9
Status: Downloaded newer image for jsgsquare/clairctl:latest
Pulling zap (owasp/zap2docker-stable)...
latest: Pulling from owasp/zap2docker-stable
423ae2b273f4: Pull complete
de83a2304fal: Pull complete
f9a83bce3af0: Pull complete
b6b53be908de: Pull complete
dfa4c0ed9f01: Pull complete
0d0271dc7f26: Pull complete

```

Ilustración 20 Despliegue ecosistema DevSecOps usando Docker-Compose vía Putty (4)

```

PuTTY (inactive)
Digest: sha256:3563ecc53448ad224262cceal85cfff8360c999c52d9c4b78630d9344dcl3cf6
Status: Downloaded newer image for owasp/zap2docker-stable:latest
Creating l8-19 absace_jenkins_1 ... done
Creating l8-19 absace_postgres_1 ... done
Creating l8-19 absace_zap_1 ... done
Creating l8-19 absace_sonarqube_1 ... done
Creating l8-19 absace_clair_1 ... done
Creating l8-19 absace_clairctl_1 ... done
Attaching to l8-19 absace_postgres_1, l8-19 absace_zap_1, l8-19 absace_clair_1, l8-19 absace_clairctl_1, l8-19 absace_sonarqube_1, l8-19 absace_jenkins_1
clair_1 | {"Event": "pgsql: could not open database: dial tcp 172.18.0.2:5432: connect: connection refused", "Level": "fatal", "Location": "main.go:96", "Time": "2020-09-04 11:14:48.423867"}
l8-19 absace_clair_1 exited with code 1
postgres_1 | The files belonging to this database system will be owned by user "postgres".
postgres_1 | This user must also own the server process.
postgres_1 |
postgres_1 | The database cluster will be initialized with locale "en_US.utf8".
postgres_1 | The default database encoding has accordingly been set to "UTF8".
postgres_1 | The default text search configuration will be set to "english".
postgres_1 |
postgres_1 | Data page checksums are disabled.
postgres_1 |
postgres_1 | fixing permissions on existing directory /var/lib/postgresql/data ... ok
postgres_1 | creating subdirectories ... ok

```

Ilustración 21 Despliegue ecosistema DevSecOps usando Docker-Compose vía Putty (5)

4.3.2 Jenkins: configuración y puesta a punto

Una vez se ha levantado el contenedor de Jenkins, se realiza la configuración inicial como si fuera una aplicación nativa de la máquina Host que lo ejecuta [67].

Para trabajar adecuadamente con las diferentes herramientas, es necesario llevar a cabo una serie de configuraciones internas de Jenkins. En los puntos siguientes, se encuentra la configuración realizada tanto en *Configure System* como en *Global Tool Configuration*, así como la elección e instalación necesaria de *plugins* para poder exprimir al servidor de Jenkins.

4.3.2.1 Plugins instalados

Para poder trabajar con las distintas herramientas de desarrollo de software como pueden ser Git, GitHub, Maven, Gradle, SonarQube, Docker, etc., añadiendo así sus funcionalidades a las de Jenkins, las cuales pueden ser ejecutar las pruebas, realizar análisis de código, enviar notificaciones mediante un correo electrónico, etc., utilizamos los diferentes *Plugins* disponibles para Jenkins. Estos son instalables directamente desde Jenkins, accediendo desde la página principal de Jenkins a *Manage Jenkins > Manage Plugins*.

Desde esta sección podremos consultar los *plugins* que tiene instalado Jenkins, así como sus posibles actualizaciones, además de una inmensa variedad disponibles para instalar ya que Jenkins dispone de un repositorio para *plugins* [68] desde el que se puede acceder a toda la información relativa al plugin, así como su descarga e instalación manual a través del apartado *Advanced*.

A continuación, se encuentra una lista de todos los *plugins* de Jenkins instalados para este Trabajo, aunque es cierto que no todos son necesarios para su correcto funcionamiento:

- Ant Plugin (version 1.11)
- Apache HttpComponents Client 4.x API Plugin (version 4.5.10-2.0)
- Authentication Tokens API Plugin (version 1.4)
- bouncycastle API Plugin (version 2.18)
- Branch API Plugin (version 2.5.9)
- Build Timeout (version 1.20)
- Command Agent Launcher Plugin (version 1.4)
- Conditional BuildStep (version 1.3.6)
- Config File Provider Plugin (version 3.6.3)
- Credentials Binding Plugin (version 1.23)
- Credentials Plugin (version 2.3.12)
- Dashboard View (version 2.13)
- Display URL API (version 2.3.3)
- Docker API Plugin (version 3.1.5.2)
- Docker Commons Plugin (version 1.17)
- Docker Pipeline (version 1.24)
- Docker plugin (version 1.2.0)
- docker-build-step (version 2.5)
- Durable Task Plugin (version 1.34)
- ECharts API Plugin (version 4.8.0-2)
- Email Extension Plugin (version 2.75)
- EnvInject API Plugin (version 1.7)
- Environment Injector Plugin (version 2.3.0)
- Folders Plugin (version 6.14)
- Git client plugin (version 3.4.2)
- Git plugin (version 4.4.1)
- GIT server Plugin (version 1.9)
- GitHub API Plugin (version 1115)
- GitHub Branch Source Plugin (version 2.8.3)
- GitHub plugin (version 1.31.0)
- Gradle Plugin (version 1.36)
- H2 API Plugin (version 1.4.199)
- HTML Publisher plugin (version 1.23)
- Jackson 2 API Plugin (version 2.11.2)
- Javadoc Plugin (version 1.6)
- JavaScript GUI Lib: ACE Editor bundle plugin (version 1.1)

- JavaScript GUI Lib: Handlebars bundle plugin (version 1.1.1)
- JavaScript GUI Lib: jQuery bundles (jQuery and jQuery UI) plugin (version 1.2.1)
- JavaScript GUI Lib: Moment.js bundle plugin (version 1.1.1)
- JQuery3 API Plugin (version 3.5.1-1)
- JSch dependency plugin (version 0.1.55.2)
- JUnit Plugin (version 1.32)
- LDAP Plugin (version 1.24)
- Lockable Resources plugin (version 2.8)
- Mailer Plugin (version 1.32)
- MapDB API Plugin (version 1.0.9.0)
- Matrix Authorization Strategy Plugin (version 2.6.2)
- Matrix Project Plugin (version 1.17)
- Maven Integration plugin (version 3.7)
- Official OWASP ZAP Jenkins Plugin (version 1.1.0)
- OkHttp Plugin (version 3.14.9)
- Oracle Java SE Development Kit Installer Plugin (version 1.4)
- OWASP Dependency-Check Plugin (version 5.1.1)
- OWASP Markup Formatter Plugin (version 2.1)
- PAM Authentication plugin (version 1.6)
- Pipeline (version 2.6)
- Pipeline Graph Analysis Plugin (version 1.10)
- Pipeline Maven Integration Plugin (version 3.8.3)
- Pipeline: API (version 2.40)
- Pipeline: Basic Steps (version 2.20)
- Pipeline: Build Step (version 2.13)
- Pipeline: Declarative (version 1.7.2)
- Pipeline: Declarative Extension Points API (version 1.7.2)
- Pipeline: GitHub Groovy Libraries (version 1.0)
- Pipeline: Groovy (version 2.82)
- Pipeline: Input Step (version 2.12)
- Pipeline: Job (version 2.39)
- Pipeline: Milestone Step (version 1.3.1)
- Pipeline: Model API (version 1.7.2)
- Pipeline: Multibranch (version 2.22)
- Pipeline: Nodes and Processes (version 2.35)
- Pipeline: REST API Plugin (version 2.14)
- Pipeline: SCM Step (version 2.11)
- Pipeline: Shared Groovy Libraries (version 2.17)
- Pipeline: Stage Step (version 2.5)
- Pipeline: Stage Tags Metadata (version 1.7.2)
- Pipeline: Stage View Plugin (version 2.14)
- Pipeline: Step API (version 2.22)
- Pipeline: Supporting APIs (version 3.5)
- Plain Credentials Plugin (version 1.7)
- Plugin Utilities API Plugin (version 1.2.5)
- Resource Disposer Plugin (version 0.14)
- Run Condition Plugin (version 1.3)
- SCM API Plugin (version 2.6.3)
- Script Security Plugin (version 1.74)
- Snakeyaml API Plugin (version 1.26.4)
- SonarQube Scanner for Jenkins (version 2.11)
- SSH Build Agents plugin (version 1.31.2)
- SSH Credentials Plugin (version 1.18.1)
- Structs Plugin (version 1.20)
- Subversion Plug-in (version 2.13.1)
- Timestamp (version 1.11.5)
- Token Macro Plugin (version 2.12)
- Trilead API Plugin (version 1.0.8)
- Workspace Cleanup Plugin (version 0.38)

4.3.2.2 Configure System

En este apartado, al cual se accede en Jenkins a través de *Manage Jenkins > Configure System*, se ha realizado la configuración en lo referente a la accesibilidad del servidor de Jenkins, así como de la integración del servidor SonarQube con Jenkins para que ambos puedan trabajar durante el flujo de ejecución del Pipeline, mediante la configuración de los parámetros oportunos que se detallan a continuación, así como imágenes del resultado de dicha configuración sobre Jenkins:

4.3.2.2.1 Maven Project Configuration

- *Global MAVEN_OPTS*: dejar vacío.
- *Location Maven Repository*: *Default (~/.m2/repository)*
- *# of executions*: 2
- *Labels*: dejar vacío.
- *Quit period*: 5
- *SCM checkout retry count*: 0

Ilustración 22. Configuración de la sección Maven Project

4.3.2.2.2 SonarQube Servers

- Activar el *checkbox* de *Enable injection Sonarqube server configuration as build environment variables*
- *Name (darle un nombre)*: *SonarQubeServer*
- *Server authentication token*: token generado en SonarQube para poder acceder y volcar los resultados obtenidos en el Pipeline cuando se realice su análisis. Se obtiene siguiendo el procedimiento descrito en el apartado *Creación del Token para SonarQube*. Para poder añadir esta credencial se debe hacer mediante el gestor de credenciales globales de Jenkins en la URL http://***:8080/credentials/store/system/domain/

The screenshot shows the Jenkins configuration page for SonarQube servers. The page is titled "Jenkins configuration" and has a sub-header "Without a resource root URL, resources will be served from the main domain with Content-Security-Policy set." Under "Global properties", there are checkboxes for "Disable deferred wipeout on this node" (unchecked), "Environment variables" (checked), and "List of variables". A table shows one variable: Name "JAVA_HOME" and Value "/var/jenkins_home/tools/hudson.model.JDK/opencv11/jdk-11.0.1/". There are "Add" and "Delete" buttons. Below this, there are checkboxes for "Prepare jobs environment" and "Tool Locations". The "SonarQube servers" section has a checkbox "Enable injection of SonarQube server configuration as build environment variables" (checked). Below it, a note says "If checked, job administrators will be able to inject a SonarQube server configuration as environment variables in the build." There is a "SonarQube installations" section with a table showing one installation: Name "SonarQubeServer", Server URL "http://sonarqube:9000", and Server authentication token "JenkinsSonarQubeToken". There are "Add" and "Delete" buttons. Below this, there is a "List of SonarQube installations" section. The "Pipeline Speed/Durability Settings" section has a dropdown menu for "Pipeline Default Speed/Durability Level" set to "None: use pipeline default (MAX_SURVIVABILITY)". There are "Save" and "Apply" buttons at the bottom.

Ilustración 23. Configuración del servidor SonarQube

4.3.2.2.3 ZAP

- **Default Host:** localhost
- **Default Port:** 8000

The screenshot shows the Jenkins configuration page for ZAP (OWASP ZAP). The page is titled "Jenkins configuration" and has a sub-header "Without a resource root URL, resources will be served from the main domain with Content-Security-Policy set." Under "Global properties", there are checkboxes for "Disable deferred wipeout on this node" (unchecked), "Environment variables" (checked), and "List of variables". A table shows one variable: Name "JAVA_HOME" and Value "/var/jenkins_home/tools/hudson.model.JDK/opencv11/jdk-11.0.1/". There are "Add" and "Delete" buttons. Below this, there are checkboxes for "Prepare jobs environment" and "Tool Locations". The "SonarQube servers" section has a checkbox "Enable injection of SonarQube server configuration as build environment variables" (checked). Below it, a note says "If checked, job administrators will be able to inject a SonarQube server configuration as environment variables in the build." There is a "SonarQube installations" section with a table showing one installation: Name "SonarQubeServer", Server URL "http://sonarqube:9000", and Server authentication token "JenkinsSonarQubeToken". There are "Add" and "Delete" buttons. Below this, there is a "List of SonarQube installations" section. The "Pipeline Speed/Durability Settings" section has a dropdown menu for "Pipeline Default Speed/Durability Level" set to "None: use pipeline default (MAX_SURVIVABILITY)". There are "Save" and "Apply" buttons at the bottom.

Ilustración 24 Configuración del servidor OWASP ZAP

4.3.2.3 Global Tool Configuration

En este apartado, al cual se accede en Jenkins a través de *Manage Jenkins > Global Tool Configuration*, se ha realizado la configuración para la integración de las distintas herramientas que necesita y queremos que Jenkins utilice para este proyecto, en todo lo que Jenkins ofrece como automatización.

En este caso, se ha realizado la configuración de las instalaciones del JDK de Java, de Git, de SonarQube Scanner, y Maven, mediante la configuración de los parámetros oportunos que se detallan a continuación, así como imágenes del resultado de dicha configuración sobre Jenkins.

4.3.2.3.1 Maven Configuration

- *Default settings provider: Use default maven settings*
- *Default global settings provider: Use default maven global settings*

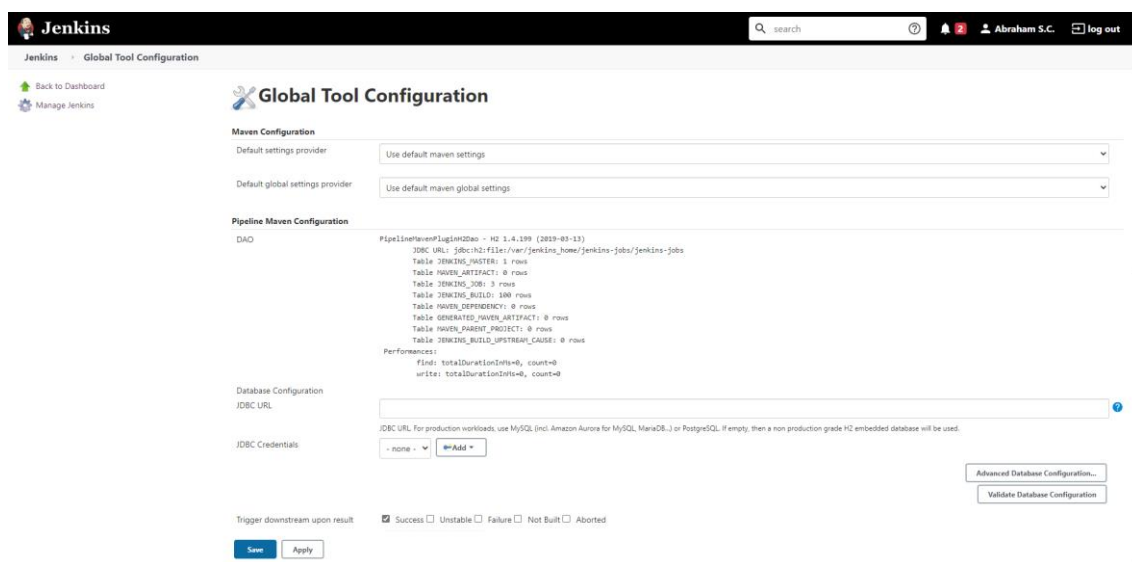


Ilustración 25. Configuración de Maven en la configuración global de Jenkins

4.3.2.3.2 JDK

Para el uso de la aplicación WebGoat es necesario tener instalado JDK 11 o superior, por lo que se debe realizar una instalación del OpenJDK11 para evitar errores de versión JAVA:

- *Name (darle un nombre): openjdk11*
- Activar el *checkbox* de *Install automatically*
- *Label*: dejar vacío
- *Download URL*: <https://download.java.net/java/GA/jdk11/13/GPL/openjdk-11.0.1-linux-x64-bin.tar.gz>
- *Subdirectory of extract*: dejar vacío.
- *Default global settings provider: Use default maven global settings*

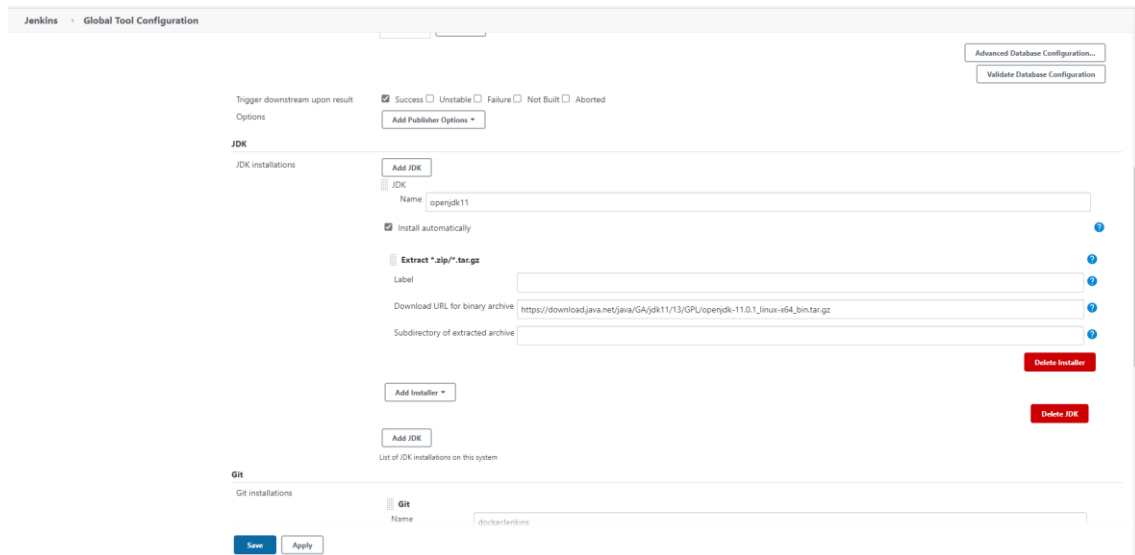


Ilustración 26 Configuración de JDK en la configuración global de Jenkins

4.3.2.3.3 SonarQube Scanner

- **Name** (darle un nombre): *SonarQubeScanner*
- Activar el *checkbox* de *Install automatically*
- *Install from Maven Central* versión: elegir la versión más conveniente. En este caso, se ha instalado la versión 4.3.0.2102.

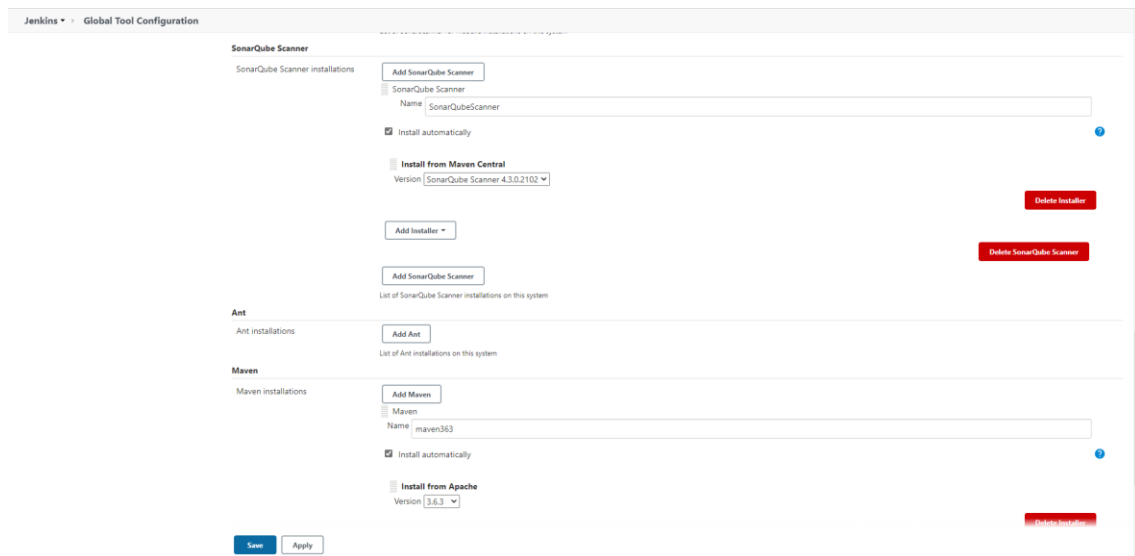


Ilustración 27. Configuración de SonarQube Scanner en la configuración global de Jenkins

4.3.2.3.4 Maven

La siguiente configuración se puede ver en la imagen del punto anterior:

- **Name** (darle un nombre): *maven363*
- Activar el *checkbox* de *Install automatically*
- *Install from Apache* versión: elegir la versión más conveniente. En este caso, se ha instalado la versión 3.6.3.

4.3.2.3.5 Git

- **Name** (darle un nombre): *Default*
- **Path to Git executable**: *git*

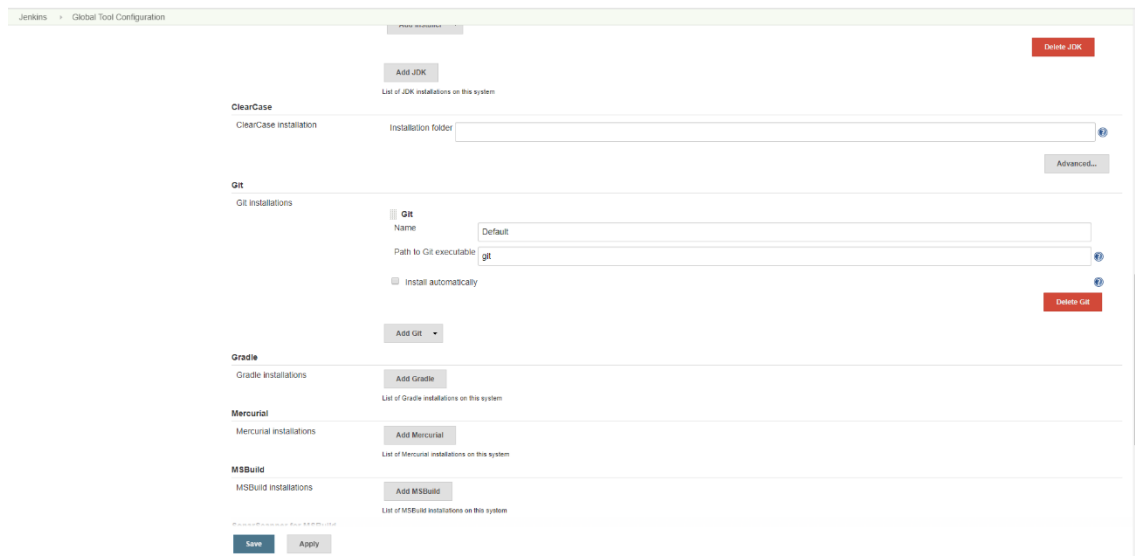


Ilustración 28. Configuración de Git en la configuración global de Jenkins

4.3.3 SonarQube: configuración y puesta a punto

Una vez se ha levantado el contenedor de SonarQube, se realiza la configuración inicial como si fuera una aplicación nativa de la máquina Host que lo ejecuta [69].

En este punto se encuentra levantado y operativo el servidor de SonarQube (como resultado de la orquestación de contenedores proporcionada por Docker-Compose) con el que trabajaremos en este proyecto:

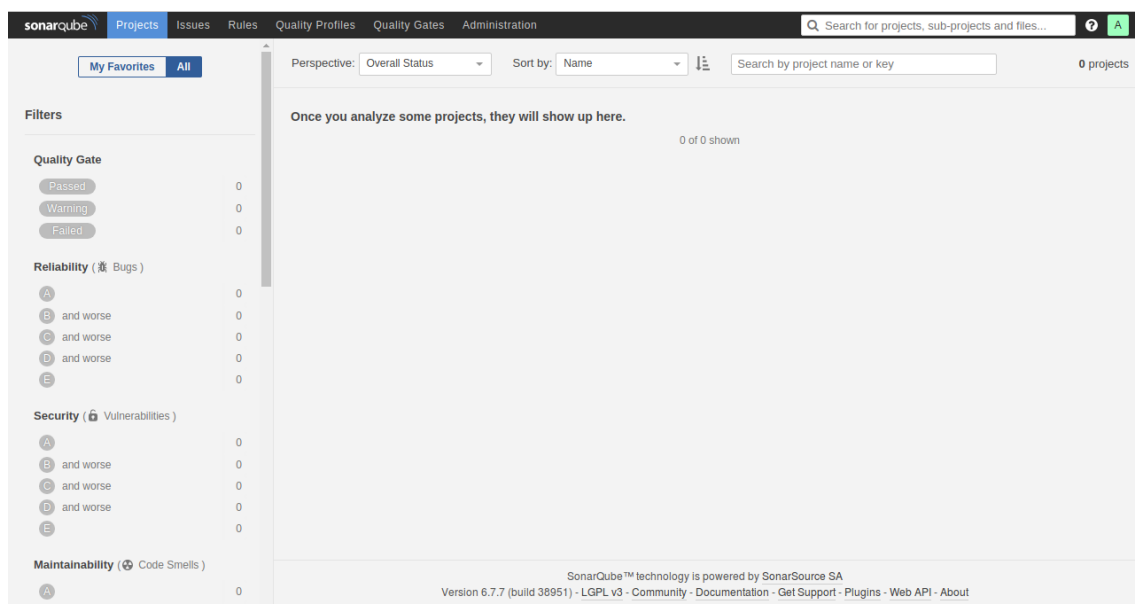


Ilustración 29. Servidor SonarQube recién configurado esperando analizar

En este momento, y por motivos de sencillez, se ha decidido que en lo referente a trabajar con SonarQube solamente se usará el usuario de administrador que viene configurado por defecto, que es usuario: *admin* y contraseña: *admin*. Este usuario será necesario para poder acceder al servidor y visualizar los reportes que realicen los análisis de SonarQube-Scanner en Jenkins.

SonarQube tiene configuradas unas políticas de seguridad por defecto, las cuales se han usado como políticas aceptadas en la etapa de Planificación.

4.3.3.1 Creación del Token para SonarQube

Una vez nos hemos logueado en el servidor, será necesario crear un token de acceso a éste para Jenkins, para ello debemos dirigirnos a la cuenta de admin, pinchando en icono de la A de color verde en la esquina superior derecha. Una vez entremos en nuestra cuenta, debemos ver algo similar a la siguiente imagen si entramos en el apartado de *Security*:

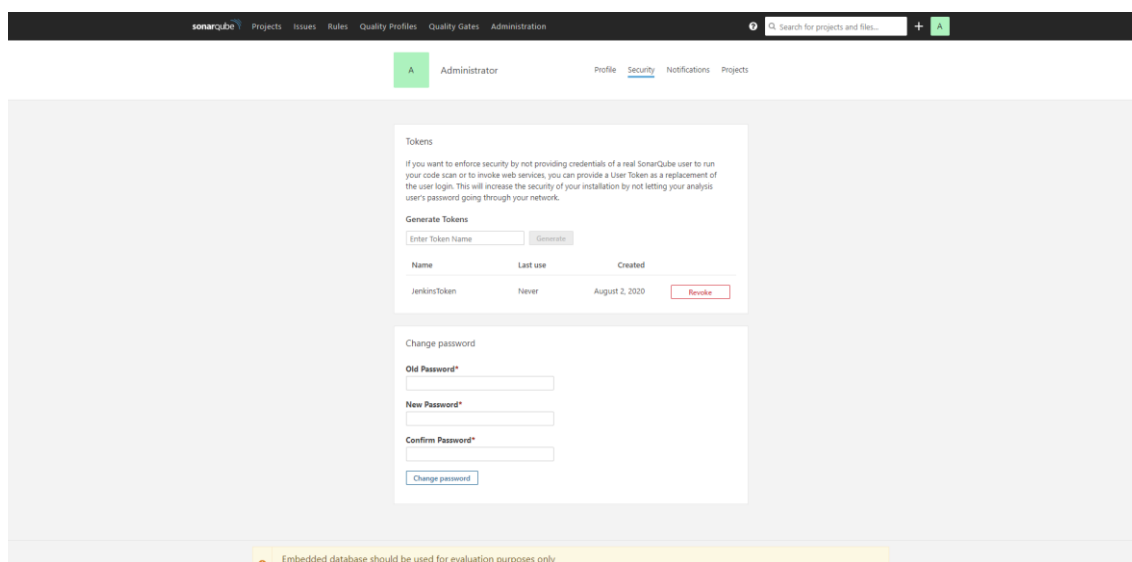


Ilustración 30. Creación del Token para SonarQube

Nos aparecerá la opción de generar un token de autenticación. De esta manera se proporciona la seguridad necesaria para autenticarse en el servidor sin necesidad de proporcionar credenciales de acceso. Se ve cómo puede crearse el Token con solo darle un nombre identificativo y pulsar en el botón *Generate*, el token aparecerá generado, y solo por esa vez, en texto plano para que podamos almacenarlo o directamente cargarlo en Jenkins.

4.4 CODIFICACIÓN Y PRUEBAS

Para la realización de este Pipeline, se ha decidido dividir su flujo en diferentes etapas, dependientes en forma de cascada descendiente, pero claramente diferencias, en la que se realizan tareas sencillas para completar el proceso de CI/CD. Esas etapas son las que se pueden ver en la siguiente imagen (como resultado de las ejecuciones del Pipeline):

Declarative: Tool Install	Checkout repository changes	SAST analysis	Dependency analysis	Docker Build	Docker Security scanner	Docker Deployment	DAST Analysis	Stop and Delete App Docker Container
58ms	702ms	1min 1s	30s	34s	15s	7s	6s	446ms
57ms	690ms	1min 0s	30s	34s	15s	10s	19s	1s
54ms	711ms	1min 0s	29s	34s	15s	10s	884ms	90ms
							failed	failed

Ilustración 31. Etapas configuradas del Pipeline DevSecOps en Jenkins

Estas etapas han sido el resultado de la realización de pruebas con diferentes versiones del script de Pipeline que finalmente ha dado los resultados buscados para cada una de las etapas, es decir, completar cada una de ellas con éxito generando bola azul (existe un *plugin* para cambiar el azul por verde, asemejando los estados a un semáforo) en el Job del Pipeline de Jenkins.

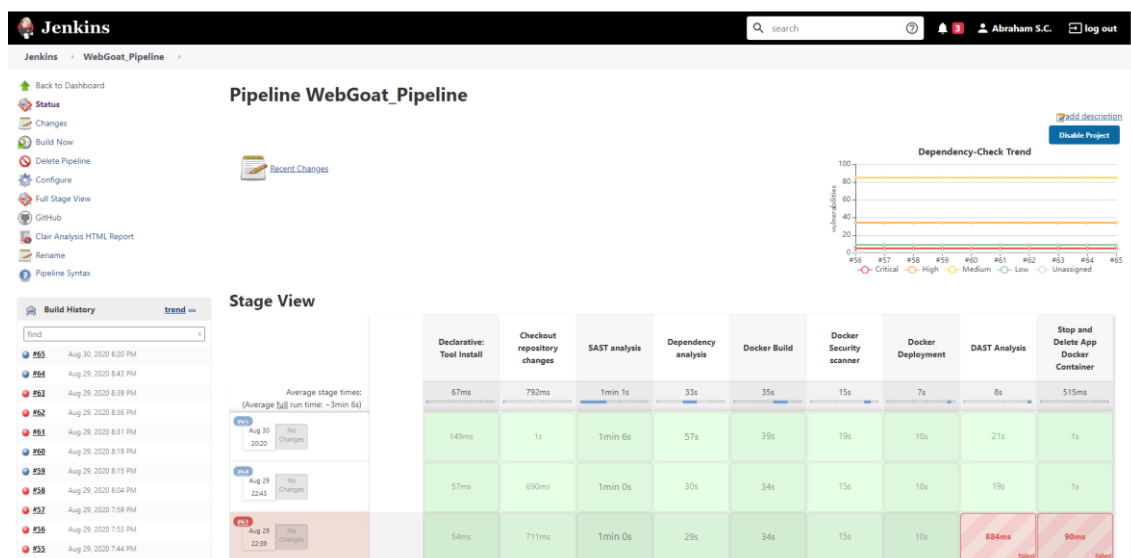


Ilustración 32. Visión global del Pipeline DevSecOps configurado y funcionando

Como puede observarse en la imagen anterior, se han realizado numerosas pruebas para obtener ejecuciones exitosas con la implementación correcta en el Pipeline.

Se ha realizado una demo del despliegue del ecosistema con *Docker-Compose*, así como una ejecución exitosa del Pipeline DevSecOps, que puede verse en: <https://youtu.be/HkFP6tAobp0>

4.4.1 Configuración del Pipeline

Se ha creado un Job de tipo Pipeline con la configuración final que se detalla a continuación, con los parámetros oportunos, además de las imágenes siguientes donde se ve el resultado de dicha configuración sobre Job del tipo Pipeline de Jenkins:

- General
 - *GitHub project*: <https://github.com/Abrin09/WebGoat> (ver apartado 5.2)
- *Build Triggers*
 - Ningún checkbox activado
- *Advance Project Options*
 - Sin configuración.
- *Pipeline*
 - *Definition: Pipeline Script* (Anexo 2: Pipeline script)

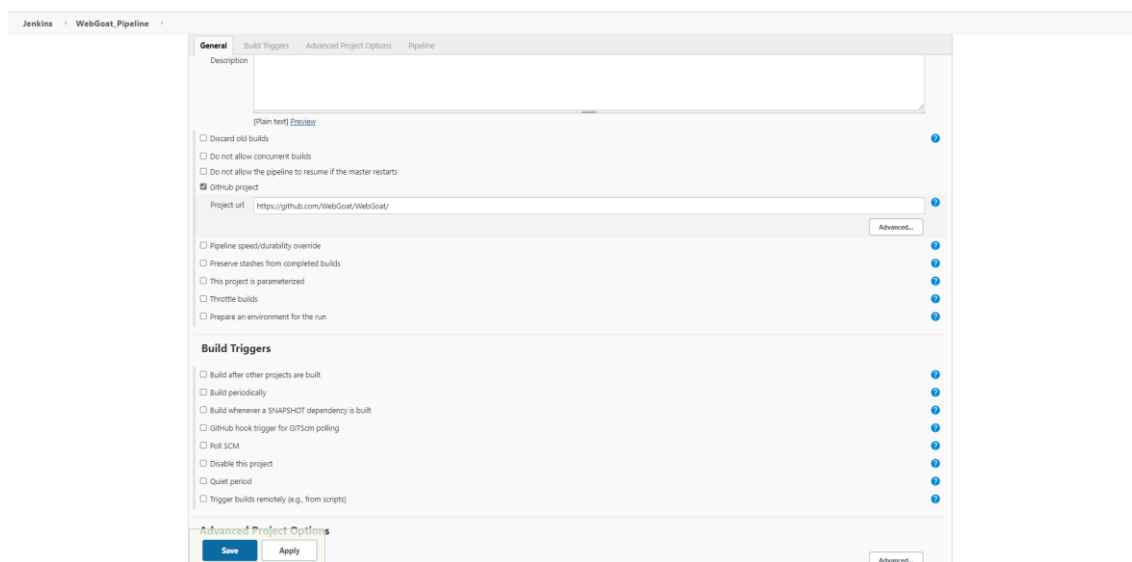


Ilustración 33. Opciones de configuración del Pipeline DevSecOps (1)

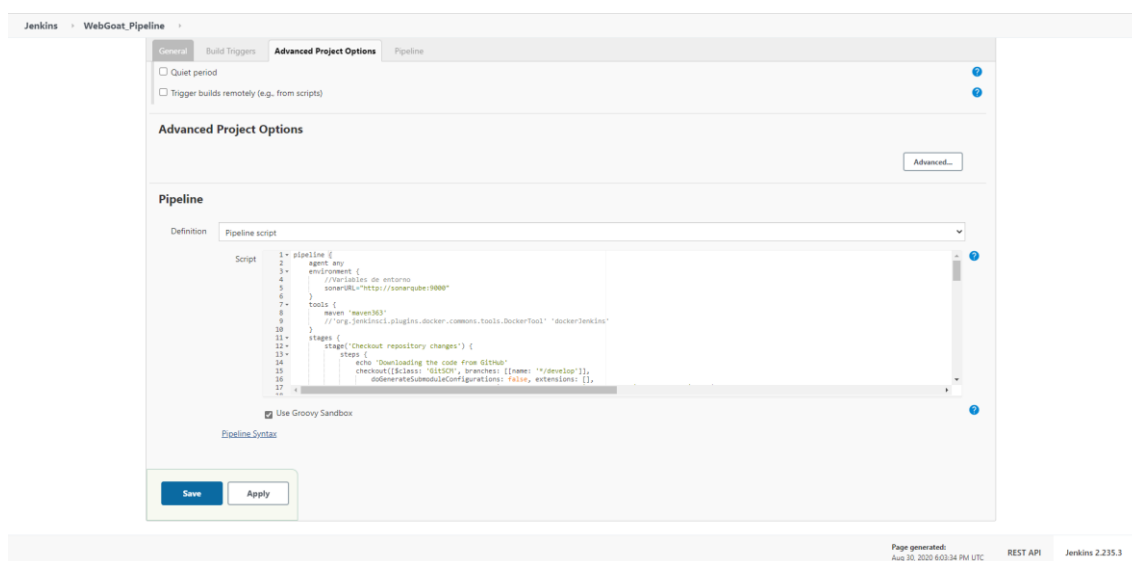


Ilustración 34. Opciones de configuración del Pipeline DevSecOps (2)

4.4.2 Etapas del Pipeline

En este apartado se pretende dar explicación a cada una de las etapas creadas en el script del Pipeline DevSecOps obtenido en este Trabajo.

4.4.2.0 Declarative: Tool Install

Esta etapa es autocreada por Jenkins debido a la configuración expuesta en el servidor y la utilización de herramientas dentro del Pipeline para sean interpretadas correctamente y no se generen errores de ejecución:

```
tools {
  maven 'maven363'
}
```

En este caso en concreto es para la herramienta Maven [18] utilizada para la creación de los binarios de la aplicación web WebGoat en la etapa 4.4.2.4 *Release and Docker Build*.

4.4.2.1 Checkout repository changes

La primera etapa del ciclo de vida DevSecOps es la monitorización del repositorio del proyecto software de la aplicación. Se debe descargar la última versión del código para poder aplicar sobre este el Pipeline DevSecOps:

```
stage('Checkout repository changes') {
  steps {
    echo 'Downloading the code from GitHub'
    checkout([$class: 'GitSCM', branches: [[name: '*/master']],
      doGenerateSubmoduleConfigurations: false, extensions: [],
      submoduleCfg: [], userRemoteConfigs: [[url:
'https://github.com/Abrin09/WebGoat.git']]])
  }
}
```

Como puede observarse, se realiza un checkout de la rama máster.

4.4.2.2 SAST analysis

Una vez disponemos en el Workspace del Job de Jenkins la última versión del código, esta etapa consiste en el análisis SAST, empleando el servidor SonarQube desplegado y configurado, a través de la herramienta SonarQube-Scanner. Es necesario especificar la URL donde se encuentra el servidor SonarQube, el nombre del proyecto que se quiera en SonarQube, la versión, y configuración adicional, como la exclusión de ficheros que no queramos analizar:

```
stage('SAST analysis') {
  steps {
    echo 'Performing SAST analysis to the code from the commit'
    sh "/var/jenkins_home/tools/hudson.plugins.sonar.SonarRunnerInstallation/SonarQubeScanner/bin/sonar-scanner -X -Dsonar.host.url=$sonarURL -Dsonar.project-Key=webgoat -Dsonar.projectName=webgoat -Dsonar.projectVersion=1.0 -Dsonar.java.bina-ries=. -Dsonar.exclusions=**/*.ts"
    echo 'SAST Finished!!!'
  }
}
```

4.4.2.3 Dependency analysis

Una vez se ha realizado el análisis SAST, esta etapa consiste en el análisis de dependencias del proyecto software haciendo uso de la herramienta OWASP Dependency Check. Este analizador

recorrerá de forma recursiva todas las carpetas del Workspace en busca del archivo de gestión de dependencias, en este caso “pom.xml”, ya que el proyecto utiliza Maven para compilar. Finalmente, se especifica la publicación del informe del análisis como un fichero XML utilizando `dependencyCheckPublisher` y así poder visualizarlo directamente en Jenkins:

```
stage('Dependency analysis') {
    steps {
        echo 'Performing Code Dependency analysis from the commit'
        dependencyCheck additionalArguments: '', odcInstallation: 'dependencyCheckOWASP'
        echo 'Finished!!!'
        dependencyCheckPublisher pattern: 'dependency-check-report.xml'
        echo 'Publishing results...'
    }
}
```

4.4.2.4 Release and Docker Build

Una vez se han pasado las etapas de análisis estático, se realiza esta etapa donde se realiza la construcción de los binarios y de la imagen Docker de la aplicación web. Como ya se ha dicho, se hace uso de Maven para compilar el código fuente utilizando `mvn clean install -DskipTests` (se evitan los test por fallo del código publicado actualmente). Posteriormente, se construye la imagen Docker definida en el fichero “Dockerfile”:

```
stage('Release and Docker Build') {
    steps {
        echo 'Compiling Code'
        sh "mvn clean install -DskipTests"
        echo 'Compiled Code Successful'
        echo 'Building docker image'
        sh "docker build -t webgoat/webgoat-8.1 ./webgoat-server"
        echo 'Docker image builded'
    }
}
```

4.4.2.5 Docker Security scanner

Obtenida una imagen Docker funcional, es momento de pasar a la etapa de análisis de vulnerabilidades en imágenes Docker, ya que se ha explicado anteriormente en este Trabajo, las imágenes de Docker son vectores vulnerables, ya sea por la imagen base o por las distintas capas que la componen. En consecuencia, se ejecuta un comando Docker para hacer uso de Clair a través de con `Clairctl` y con `Clair-Scanner`.

Como puede observarse, se le indica a `Clairctl` cuál es la imagen para analizar, para generar después un informe HTML del mismo. Se utiliza Docker para traer dicho informe al Workspace y publicarlo.

En el caso de `Clair-Scanner`, se obtiene la herramienta al Workspace para poder hacer uso de ella dándole permisos de ejecución e indicando la dirección IP del demonio Docker para poder analizar la imagen además de indicar la dirección de Clair:

```
stage('Docker Security scanner') {
    steps {
        echo 'To Do integration of Clair with Pipeline'
        sh "docker exec -it 18-19_absace_clairctl_1 ls -la /reports/html || exit 0"
        sh "docker exec 18-19_absace_clairctl_1 clairctl analyze -l webgoat/webgoat-8.1"
        || exit 0
        sh "docker exec 18-19_absace_clairctl_1 clairctl report -l webgoat/webgoat-8.1"
        || exit 0
        sh "mkdir -p clairctl-reports"
```

```

sh "docker cp 18-19_absace_clairctl_1:/reports/html/analysis-webgoat-webgoat-
8.1-latest.html ${WORKSPACE}/clairctl-reports/analysis-webgoat-webgoat-8.1-latest.html"
sh '''
    DOCKER_GATEWAY=$(ip r | tail -n1 | awk '{ print $9 }')
    wget -qO clair-scanner https://github.com/arminc/clair-scanner/re-
leases/download/v12/clair-scanner_linux_amd64 && chmod +x clair-scanner
    ./clair-scanner --ip="$DOCKER_GATEWAY" --clair=http://clair:6060
webgoat/webgoat-8.1 || exit 0
'''
echo 'Publishing the HTML Report'
sh "mkdir -p Clair-Analysis-Report"
publishHTML([allowMissing: false, alwaysLinkToLastBuild: false, keepAll: false,
reportDir: 'clairctl-reports',
reportFiles: 'analysis-webgoat-webgoat-8.1-latest.html', reportName:
'Clair Analysis HTML Report', reportTitles: 'Clair Analysis Report'])
echo 'Publishing succesfully done'
}
}

```

4.4.2.6 Docker Deployment

Esta etapa es el despliegue automatizado de la imagen Docker obtenida, haciendo que el sistema también se convierta en un entorno de Despliegue Continuo (CD). Para ello, se arranca el contenedor, al que se la ha dado el nombre de `webgoat-asc`, a partir de la imagen creada, indicando el puerto donde se quiere desplegar la aplicación. En este caso es el puerto 8888, para que no haya conflictos con los demás utilizados.

```

stage('Docker Deployment') {
    steps {
        echo 'Deploying the application'
        sh "docker run -d --name webgoat-asc -p 8888:8080 --net 18-19_absace_mynet
webgoat/webgoat-8.1 || exit 0"
        echo 'http://webgoat-asc:8888/WebGoat'
        sleep 10
    }
}

```

4.4.2.7 DAST Analysis

Con la aplicación ya desplegada, esta etapa entra en juego, donde se realiza el análisis DAST utilizando ZAP. Haciendo uso del comando Docker se ejecuta un análisis básico de OWASP ZAP indicando la URL de la aplicación web desplegada:

```

stage('DAST Analysis') {
    steps {
        echo 'DAST Analysis Starting'
        sh "docker exec 18-19_absace_zap_1 zap-cli quick-scan --self-contained --start-
options '-config api.disablekey=true' http://webgoat-asc:8888/WebGoat"
    }
}

```

4.4.2.8 Stop and Delete App Docker Container

Esta etapa final es para detener el contenedor la aplicación web desplegada localmente, así como la eliminación de este para la siguiente ejecución del Pipeline, ya que, si no se realiza esta etapa, la siguiente ejecución del Job dará error al existir un contenedor asociado a la aplicación:

```

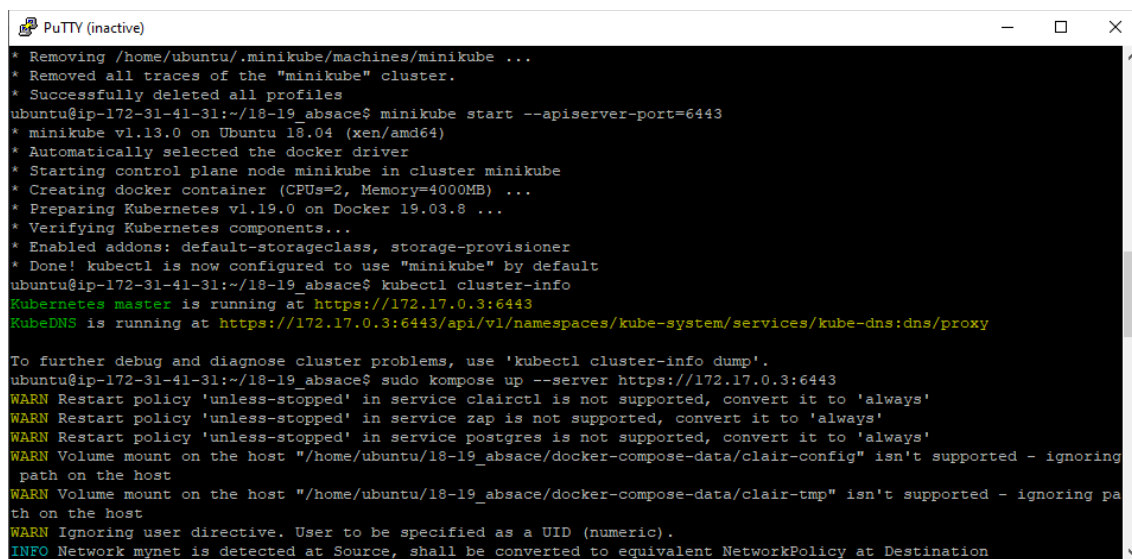
stage('Stop and Delete App Docker Container') {
    steps {
        echo 'Stopping and Deleting the App Docker Container'
        sh "docker stop webgoat-asc || exit 0"
        sh "docker rm -f webgoat-asc || exit 0"
    }
}

```

4.4.3 Migración a Kubernetes con Kompose

La migración del ecosistema DevSecOps hacia Kubernetes se ha planificado probado haciendo uso de la herramienta Kompose.

En el repositorio GitHub de esta herramienta [59] se encuentra un apartado de documentación, donde se explica desde como instalar la herramienta hasta empezar a usar la misma. Por ello, se ha hecho uso de esas indicaciones, intentado crear un ecosistema Kubernetes funcional y así poder hacer uso de Kompose para convertir toda la configuración exitosamente conseguida en *Docker-Compose*, tal y como se puede ver a continuación:



```
Putty (inactive)
* Removing /home/ubuntu/.minikube/machines/minikube ...
* Removed all traces of the "minikube" cluster.
* Successfully deleted all profiles
ubuntu@ip-172-31-41-31:~/18-19_absace$ minikube start --apiserver-port=6443
* minikube v1.13.0 on Ubuntu 18.04 (xen/amd64)
* Automatically selected the docker driver
* Starting control plane node minikube in cluster minikube
* Creating docker container (CPUs=2, Memory=4000MB) ...
* Preparing Kubernetes v1.19.0 on Docker 19.03.8 ...
* Verifying Kubernetes components...
* Enabled addons: default-storageclass, storage-provisioner
* Done! kubectrl is now configured to use "minikube" by default
ubuntu@ip-172-31-41-31:~/18-19_absace$ kubectrl cluster-info
Kubernetes master is running at https://172.17.0.3:6443
KubeDNS is running at https://172.17.0.3:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectrl cluster-info dump'.
ubuntu@ip-172-31-41-31:~/18-19_absace$ sudo kompose up --server https://172.17.0.3:6443
WARN Restart policy 'unless-stopped' in service clairctl is not supported, convert it to 'always'
WARN Restart policy 'unless-stopped' in service zap is not supported, convert it to 'always'
WARN Restart policy 'unless-stopped' in service postgres is not supported, convert it to 'always'
WARN Volume mount on the host "/home/ubuntu/18-19_absace/docker-compose-data/clair-config" isn't supported - ignoring path on the host
WARN Volume mount on the host "/home/ubuntu/18-19_absace/docker-compose-data/clair-tmp" isn't supported - ignoring path on the host
WARN Ignoring user directive. User to be specified as a UID (numeric).
INFO Network mynet is detected at Source, shall be converted to equivalent NetworkPolicy at Destination
```

Ilustración 35 Configuración y despliegue de sistema Kubernetes con Kompose

5 RESULTADOS OBTENIDOS

5.1 ENTORNO DOCKER

Se ha desarrollado un Pipeline DevSecOps implementando herramientas de seguridad para el Análisis Estático de Código, y de Vulnerabilidades, durante todo el proceso de Integración Continua (CI) y Despliegue Continuo (CD), siendo el resultado exitoso.

Hay que destacar que, en la implementación del script del Pipeline, las direcciones URL tienen el nombre del servicio implementado en el esquema YAML de *Docker-Compose* ya que el *Docker Daemon* de la máquina Host es capaz de resolver esas URLs correctamente. A continuación, se detallan los resultados obtenidos en cada etapa del Pipeline DevSecOps.

5.1.1 Declarative: Tool Install

Esta etapa es exitosa siempre que se haya declarado correctamente las variables o herramientas a utilizar en el Pipeline:

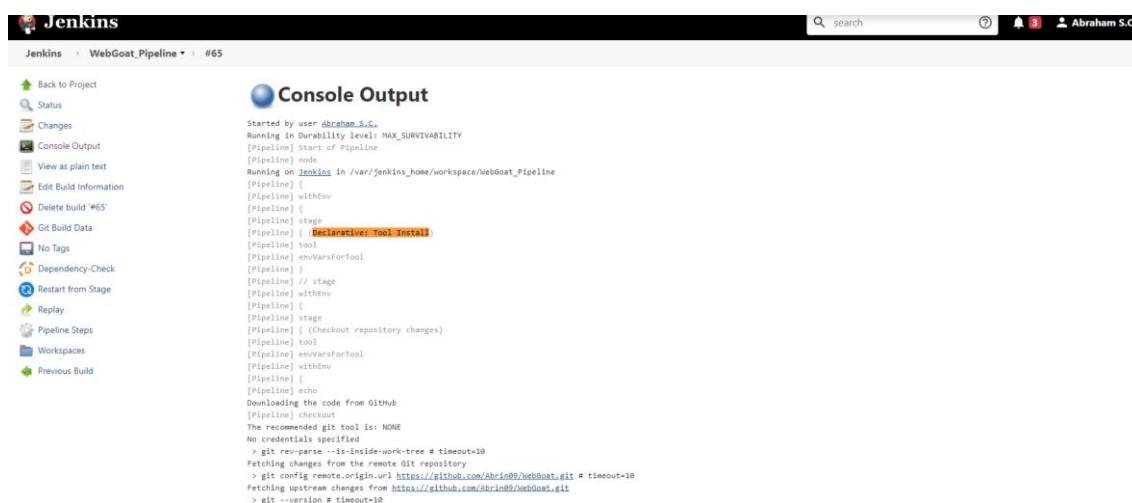


Ilustración 36 Éxito en etapa “Declarative: Tool Install”

5.1.2 Checkout repository changes

Esta etapa es exitosa siempre que se haya declarado correctamente el repositorio del código fuente, así como las credenciales que fueran necesarias para acceder a él:

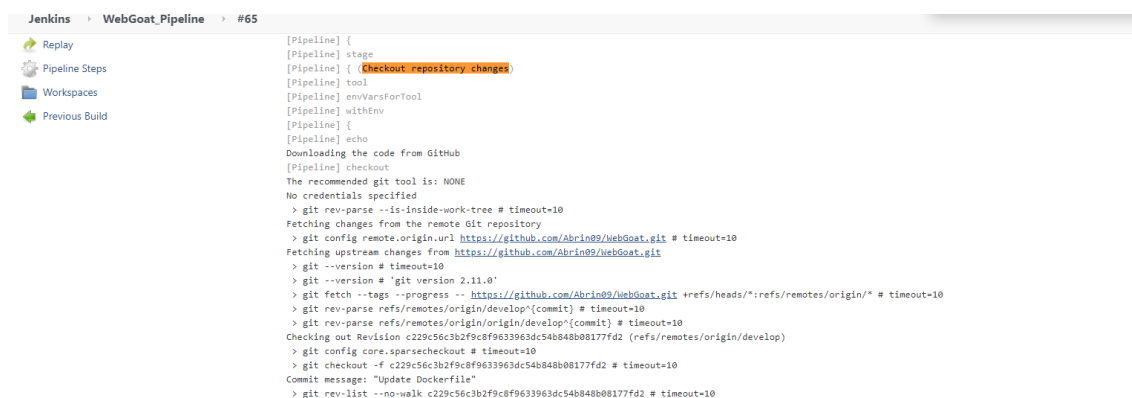


Ilustración 37 Éxito en etapa “Checkout repository changes”

5.1.3 SAST analysis

Una vez se hace la clonación del repositorio en la etapa anterior, se pasa a ejecutar esta con el análisis SAST. Como puede observarse en las imágenes siguientes, el análisis se realiza correctamente. Una vez que ha finalizado esta etapa, todos los datos referentes a este análisis se encuentran disponibles en el servidor dedicado SonarQube desplegado:

```
Jenkins  WebGoat Pipeline  #65

[Pipeline] stage
[Pipeline] {
[Pipeline] tool
[Pipeline] envvarsForTool
[Pipeline] withEnv
[Pipeline] {
[Pipeline] echo
Performing SAST analysis to the code from the commit
[Pipeline] sh
+ /var/jenkins_home/tools/hudson.plugins.sonar.SonarRunnerInstallation/SonarQubeScanner/bin/sonar-scanner -X -Dsonar.host.url=http://sonarqube:9000 -Dsonar.projectkey=webgoat -Dsonar.projectname=webgoat -
Dsonar.projectversion=1.0 -Dsonar.java.binaries=. -Dsonar.exclusions=**/*.ts
18:21:03.422 INFO: Scanner configuration file: /var/jenkins_home/tools/hudson.plugins.sonar.SonarRunnerInstallation/SonarQubeScanner/conf/sonar-scanner.properties
18:21:03.426 INFO: Project root configuration file: NONE
18:21:03.471 INFO: SonarScanner 4.3.0.2182
18:21:03.472 INFO: Java 11.0.1 Oracle Corporation (64-bit)
18:21:03.472 INFO: Linux 5.3.0-103-amd64
18:21:03.722 DEBUG: keyStore is :
18:21:03.722 DEBUG: keyStore type is : pkcs12
18:21:03.722 DEBUG: keyStore provider is :
18:21:03.723 DEBUG: init keyStore
18:21:03.723 DEBUG: Init keymanager of type SunX509
18:21:03.981 DEBUG: Create: /root/.sonar/cache
18:21:03.983 DEBUG: User cache: /root/.sonar/cache
18:21:03.983 DEBUG: Create: /root/.sonar/cache/tmp
18:21:03.986 DEBUG: Extract sonar-scanner-api-batch in temp...
18:21:03.991 DEBUG: Get bootstrap index...
18:21:03.992 DEBUG: Download: https://sonarqube:9000/bootstrap/index
18:21:03.998 DEBUG: Get bootstrap completed
18:21:03.992 DEBUG: Create isolated classloader...
18:21:04.000 DEBUG: Start temp cleaning...
18:21:04.009 DEBUG: Temp cleaning done
18:21:04.010 INFO: Scanner configuration file: /var/jenkins_home/tools/hudson.plugins.sonar.SonarRunnerInstallation/SonarQubeScanner/conf/sonar-scanner.properties
```

Ilustración 38 Éxito en etapa “SAST analysis (1)”

```
Jenkins  WebGoat Pipeline  #65

18:22:06.242 DEBUG: Detection of duplications for /var/jenkins_home/workspace/webgoat_Pipeline/webgoat-lessons/src/main/java/org/owasp/webgoat/jwt/JWTSecretKeyEndpoint.java
18:22:06.242 DEBUG: Detection of duplications for /var/jenkins_home/workspace/webgoat_Pipeline/webgoat-lessons/client-side-filtering/src/main/resources/js/clientSideFilteringFree.js
18:22:06.243 DEBUG: Detection of duplications for /var/jenkins_home/workspace/webgoat_Pipeline/webgoat-lessons/container/src/main/resources/static/js/goatApp/controller/MenuController.js
18:22:06.244 DEBUG: Detection of duplications for /var/jenkins_home/workspace/webgoat_Pipeline/webgoat-lessons/auth-bypass/src/test/org/owasp/webgoat/auth_bypass/BypassVerificationTest.java
18:22:06.244 DEBUG: Detection of duplications for /var/jenkins_home/workspace/webgoat_Pipeline/webgoat-container/src/main/resources/static/js/goatApp/controller/LessonController.js
18:22:06.244 INFO: CPD calculation finished
18:22:06.692 INFO: Analysis report generated in 396ms, dir size=16 KB
18:22:06.718 INFO: Analysis report compressed in 1625ms, zip size=5 KB
18:22:06.718 INFO: Analysis report generated in /var/jenkins_home/workspace/webgoat_Pipeline/scannerwork/scanner-report
18:22:06.718 DEBUG: Upload report
18:22:06.730 DEBUG: POST 200 http://sonarqube:9000/api/ci/submit?projectKey=webgoat&sonar.projectname=webgoat | time=244ms
18:22:06.730 INFO: Analysis report uploaded in 244ms
18:22:06.730 INFO: ANALYSIS SUCCESSFUL, you can browse https://sonarqube:9000/dashboard?id=webgoat
18:22:06.730 INFO: Note that you will be able to access the updated dashboard once the server has processed the submitted analysis report
18:22:06.730 INFO: More about the report processing at https://sonarqube:9000/api/ci/task?id=XXXXXXXXXXXXXXXXXXXX
18:22:06.730 DEBUG: Report metadata written to /var/jenkins_home/workspace/webgoat_Pipeline/scannerwork/report-task.txt
18:22:06.730 DEBUG: Post-Jobs :
18:22:06.730 INFO: Analysis total time: 1:03.794 s
18:22:06.730 INFO: -----
18:22:06.730 INFO: EXECUTION SUCCESS
18:22:06.730 INFO: -----
18:22:06.730 INFO: Total time: 1:09.341s
18:22:06.730 INFO: Final Memory: 10M/60M
18:22:06.730 INFO: -----
[Pipeline] echo
SAST finished!!!
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] {
[Pipeline] tool
[Pipeline] envvarsForTool
[Pipeline] withEnv
[Pipeline] {
[Pipeline] echo
Performing Code Dependency analysis from the commit
[Pipeline] dependencyCheck
[DependencyCheck] [INFO] Checking for updates
[DependencyCheck] [INFO] Download Started for NVD CVE - Modified
[DependencyCheck] [INFO] Download Complete for NVD CVE - Modified (1726 ms)
```

Ilustración 39 Éxito en etapa “SAST analysis (1)”

Si nos dirigimos al servidor de SonarQube, podemos observar que se encuentra el análisis al completo, con todo el detalle referente a vulnerabilidades, errores, *Code Smell*, deuda técnica, etc.

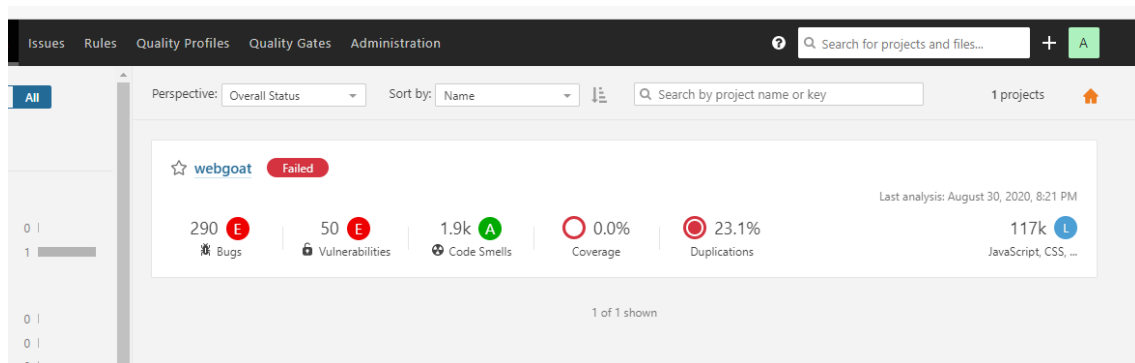


Ilustración 40 Proyecto WebGoat en SonarQube

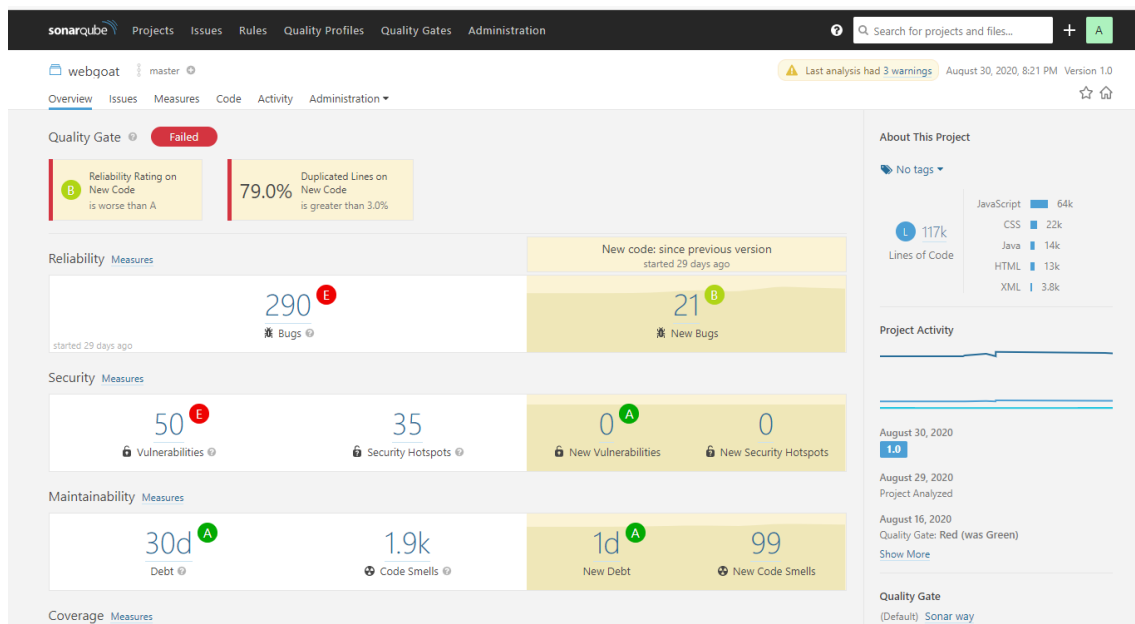


Ilustración 41 Información general del análisis sobre el proyecto WebGoat

Como puede observarse, el último análisis de SonarQube realizado ha detectado las siguientes deficiencias. También hace una comparación con los análisis anteriormente hechos, indicando las diferencias:

	Resultados en el último análisis realizado	Incremento de resultados desde el análisis anterior
Errores de Código	290	21
Vulnerabilidades de Seguridad	50	0
Code Smells	1.900	99
Deuda Técnica	30	1

A continuación, se puede observar algunas de las vulnerabilidades, errores, *Code Smell*, detectadas en este proyecto software:

The screenshot displays the SonarQube interface for the 'webgoat' project. The left sidebar contains filters for 'Type' (Vulnerability), 'Severity' (Blocker), and 'Security Category' (SonarSource). The main panel shows a list of 10 vulnerabilities, all of type 'Vulnerability' and severity 'Blocker'. Each entry includes a file path, a description, a 'See Rule' link, and a '1d 6h effort' estimate.

Ilustración 42 Vulnerabilidades bloqueantes en el proyecto WebGoat

The screenshot displays the SonarQube interface for the 'webgoat' project, showing a detailed view of a vulnerability. The left sidebar contains filters for 'Type' (Vulnerability) and 'Severity' (Blocker). The main panel shows the source code of the 'LabelServiceTest' class, with a red box highlighting the 'password' field in the '@WithMockUser' annotation.

Ilustración 43 Detalle de una vulnerabilidad bloqueante del proyecto WebGoat

5.1.4 Dependency analysis

Una vez se hace realizado el análisis SAST, se pasa a ejecutar esta etapa del análisis de dependencias del proyecto software:

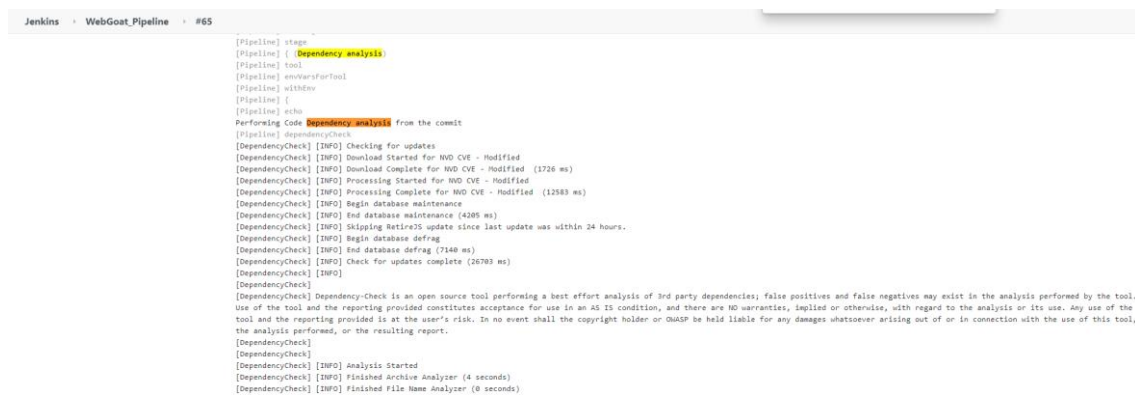


Ilustración 44 Éxito en etapa “Dependency analysis”

Una vez que ha finalizado esta etapa, todos los datos referentes a este análisis se encuentran disponibles en el Workspace del Job, accesibles en la parte izquierda de la *build* del Job. Se puede ver en detalle los resultados obtenidos en este análisis:

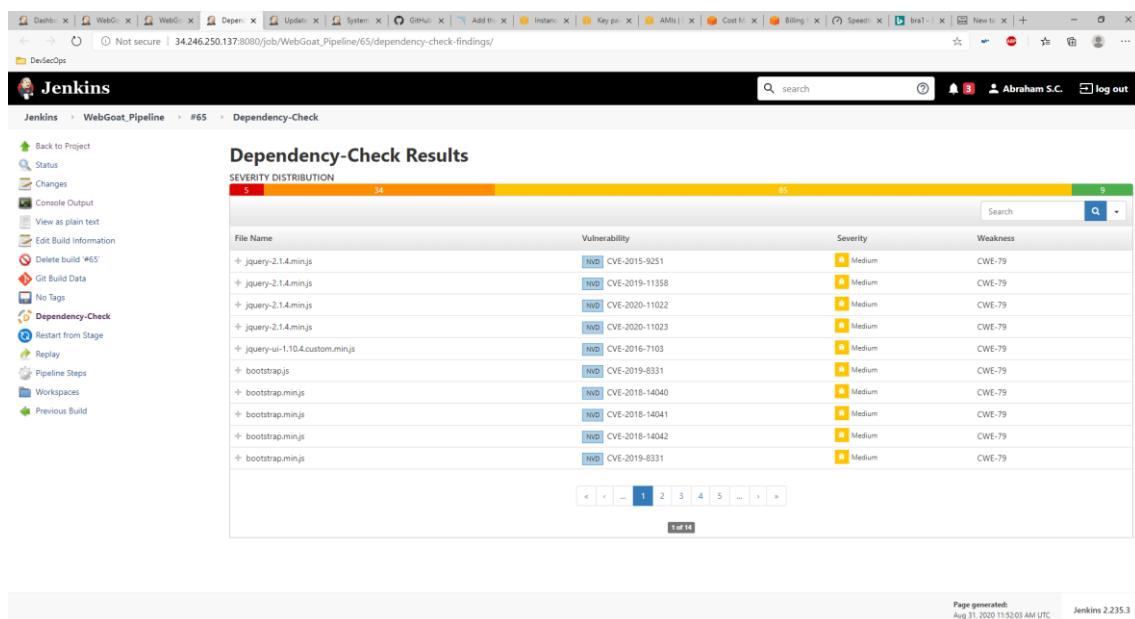


Ilustración 45 Resultados de OWASP Dependency Check (1)

OWASP Dependency Check muestra bastante información acerca de las vulnerabilidades, desde el CWE [70] o CVE [28] que identifica la vulnerabilidad, a la librería y la vulnerabilidad a la que es vulnerable. Se observa que ha detectado: 5 críticas, 34 altas, 85 medias, y 9 bajas:

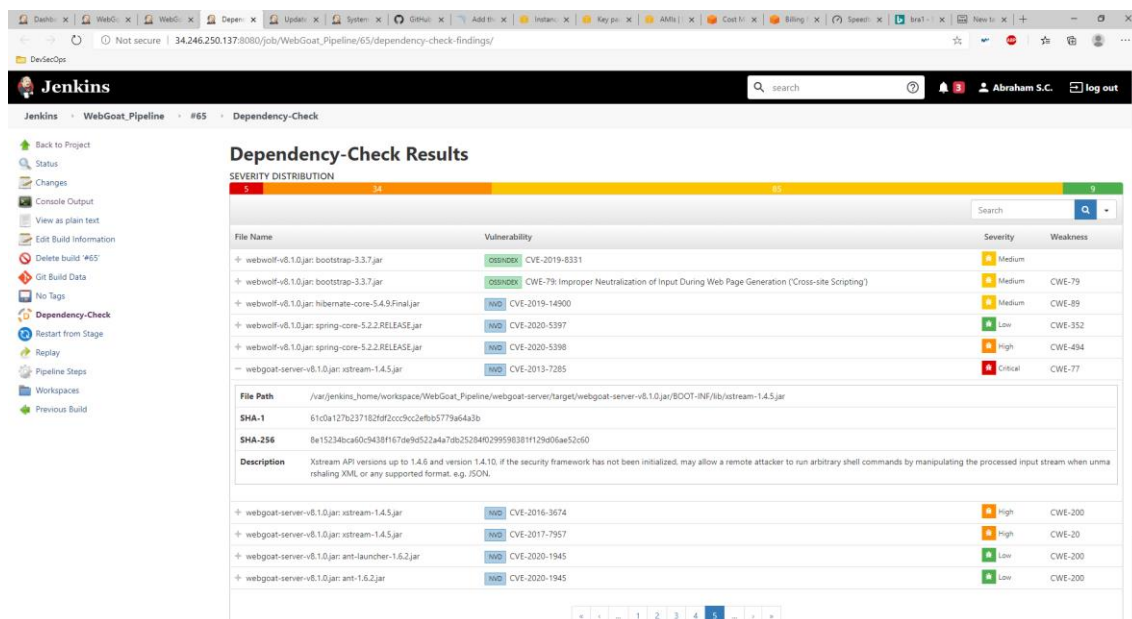


Ilustración 46 Resultados de OWASP Dependency Check (2)

5.1.5 Release and Docker Build

En esta etapa se generan los binarios correctamente haciendo uso de Maven, junto con la creación exitosa de la imagen Docker a partir de dichos binarios:



Ilustración 47 Éxito creando binarios en etapa "Release and Docker Build"

```

Jenkins  WebGoat_Pipeline  #65

+ Docker build -t webgoat/webgoat-8.1 ./webgoat-server
Sending build context to Docker daemon  87.98MB

Step 1/10 : FROM openjdk:11.0.1-jre-slim-stretch
--> 49b31a72a85a
Step 2/10 : ARG webgoat_version=v8.1.0
--> Using cache
--> 890d00f41b6d
Step 3/10 : RUN apt-get update && apt-get install && useradd --home-dir /home/webgoat --create-home -U webgoat
--> Using cache
--> f3b64e12449d
Step 4/10 : USER webgoat
--> Using cache
--> 908e555ca8a9
Step 5/10 : RUN cd /home/webgoat/; mkdir -p .webgoat-${webgoat_version}
--> Using cache
--> ba097e71831d
Step 6/10 : COPY target/webgoat-server-${webgoat_version}.jar /home/webgoat/webgoat.jar
--> d8834e177289
Step 7/10 : EXPOSE 8080
--> Running in 8affcb82c507
Removing intermediate container 8affcb82c507
--> e663e9a9446b
Step 8/10 : WORKDIR /home/webgoat
--> Running in dde3a4d841f3
Removing intermediate container dde3a4d841f3
--> 87f7f5464913
Step 9/10 : ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/./urandom", "-jar", "/home/webgoat/webgoat.jar"]
--> Running in 3e93b6532e6e
Removing intermediate container 3e93b6532e6e
--> c6c9fa26095a
Step 10/10 : CMD ["--server.port=8080", "--server.address=0.0.0.0"]
--> Running in 7fb23af8a08f
Removing intermediate container 7fb23af8a08f
--> 123057c02dc5
Successfully built 123057c02dc5
Successfully tagged webgoat/webgoat-8.1:latest
[Pipeline] echo
Docker image buildied
[Pipeline] }

```

Ilustración 48 Éxito creando imagen Docker en etapa “Release and Docker Build”

5.1.6 Docker Security scanner

En esta etapa se realiza exitosamente un análisis de vulnerabilidades sobre la imagen Docker obtenida en la etapa anterior, haciendo uso de Clairctl y Clair-Scanner como clientes de integración de Clair:

```

Jenkins  WebGoat_Pipeline  #65

[Pipeline] { (Docker Security scanner)
[Pipeline] tool
[Pipeline] envVarsForTool
[Pipeline] withEnv
[Pipeline] {
[Pipeline] echo
To Do integration of Clair with Pipeline
[Pipeline] sh
+ docker exec -it 18-19_absace_clairctl_1 ls -la /reports/html
the input device is not a TTY
+ exit 0
[Pipeline] sh
+ docker exec 18-19_absace_clairctl_1 clairctl analyze -l webgoat/webgoat-8.1

Image: /webgoat/webgoat-8.1:latest
9 layers found

→ Analysis [3e96f54948f8] found 173 vulnerabilities.
→ Analysis [21e9982ef9d1] found 173 vulnerabilities.
→ Analysis [affa2b3dead0] found 173 vulnerabilities.
→ Analysis [f5c5bd861ba1] found 173 vulnerabilities.
→ Analysis [a6175e7e36da] found 88 vulnerabilities.
→ Analysis [8ac6162c2dc5] found 88 vulnerabilities.
→ Analysis [55554b21af5a] found 88 vulnerabilities.
→ Analysis [6344b42c1c8e] found 88 vulnerabilities.
→ Analysis [08bf86d66244] found 86 vulnerabilities.

cleaning temporary local repository: remove /tmp: device or resource busy
+ exit 0
[Pipeline] sh
+ docker exec 18-19_absace_clairctl_1 clairctl report -l webgoat/webgoat-8.1
HTML report at /reports/html/analysis-webgoat-webgoat-8.1-latest.html
cleaning temporary local repository: remove /tmp: device or resource busy
+ exit 0
[Pipeline] sh

```

Ilustración 49 Éxito utilizando Clairctl en etapa “Docker security scanner”


```
Jenkins > WebGoat_Pipeline > #65

+ chmod +x clair-scanner
+ ./clair-scanner --ip=172.18.0.4 --clair=https://clair:6666 webgoat/webgoat-8.1
2020/08/30 18:24:01 [0:32m[INFO] # Start clair-scanner[0m
2020/08/30 18:24:04 [0:32m[INFO] # Server listening on port 9279[0m
2020/08/30 18:24:04 [0:32m[INFO] # Analyzing 08f8666c3456e4c70d10971224c300030770848f8c5b2b07d72e3f6869b28[0m
2020/08/30 18:24:04 [0:32m[INFO] # Analyzing 6344b42c1c8e3c1134fa2fde0e114199376a6f380c41a40426ca90cb26d[0m
2020/08/30 18:24:04 [0:32m[INFO] # Analyzing 55554b21ef9ae92d6b70a8767eae58bafe5a34e159858f107103218966000c[0m
2020/08/30 18:24:04 [0:32m[INFO] # Analyzing 6aca162c2dcdf2c2dc6a3983b4584e9b09802ff8f6530e733f828438c3827ae9[0m
2020/08/30 18:24:04 [0:32m[INFO] # Analyzing a6175e7e3da80235b11eb57b283c1848db284e9656354fe18e42263a67f[0m
2020/08/30 18:24:04 [0:32m[INFO] # Analyzing fc5cb0861ba1bdef0682f3821c0424935483e9813e1e5810029750752a19ae[0m
2020/08/30 18:24:04 [0:32m[INFO] # Analyzing affa3b36a0019d158f9b31063f808f04dc1a2d71a45e1f412a2930a10e[0m
2020/08/30 18:24:04 [0:32m[INFO] # Analyzing 21e9982ef9d15e421de49186be3b075f6c300a991c9b763113c7b1b5585c21[0m
2020/08/30 18:24:04 [0:32m[INFO] # Analyzing 3e96f54948f82241d6eb123847ba69edbe9b2967726ef123b5457b4b4ef7e359[0m
2020/08/30 18:24:04 [0:32m[WARN] # Image [webgoat/webgoat-8.1] contains 173 total vulnerabilities[0m
2020/08/30 18:24:04 [0:31m[ERROR] # Image [webgoat/webgoat-8.1] contains 173 unapproved vulnerabilities[0m

+-----+-----+-----+-----+-----+
| STATUS | CVE SEVERITY | PACKAGE NAME | PACKAGE VERSION | CVE DESCRIPTION |
+-----+-----+-----+-----+-----+
| [1:31mUnapproved[0m | Low CVE-2017-15804 | glibc | 2.24-11+deb9u3 | The glob function in glob.c in the GNU C Library (aka | | | | |
| | | | | | | | | glibc or libc) before 2.27 contains a buffer overflow |
| | | | | | | | | during unescaping of user names with the ~ operator. |
| | | | | | | | | https://security-tracker.debian.org/tracker/CVE-2017-15804 |
+-----+-----+-----+-----+-----+
| [1:31mUnapproved[0m | Low CVE-2019-17594 | ncurses | 6.0+20161126-1+deb9u2 | There is a heap-based buffer over-read in the | | | | |
| | | | | | | | | _nc_find_entry function in tinfo/comp_hash.c in the |
| | | | | | | | | terminfo library in ncurses before 6.1-20191012. |
| | | | | | | | | https://security-tracker.debian.org/tracker/CVE-2019-17594 |
+-----+-----+-----+-----+-----+
| [1:31mUnapproved[0m | Low CVE-2019-17595 | ncurses | 6.0+20161126-1+deb9u2 | There is a heap-based buffer over-read in the | | | | |
| | | | | | | | | _nc_find_entry function in tinfo/comp_hash.c in the |
| | | | | | | | | terminfo library in ncurses before 6.1-20191012. |
| | | | | | | | | https://security-tracker.debian.org/tracker/CVE-2019-17595 |
+-----+-----+-----+-----+-----+
| [1:31mUnapproved[0m | Low CVE-2018-12384 | nss | 2:3.26-2-1+deb9u1 | When handling a SSLv2-compatible ClientHello request, the |
| | | | | | | | | server doesn't generate a new random value but sends an |
```

Ilustración 50 Éxito utilizando Clair-Scanner en etapa “Docker security scanner”

Los resultados nos indican que se trata de una imagen muy vulnerable con 173 vulnerabilidades en sus 9 layers. La información que nos proporciona Clair es el CVE detectado y la layer afectada, además de una descripción de la vulnerabilidad:

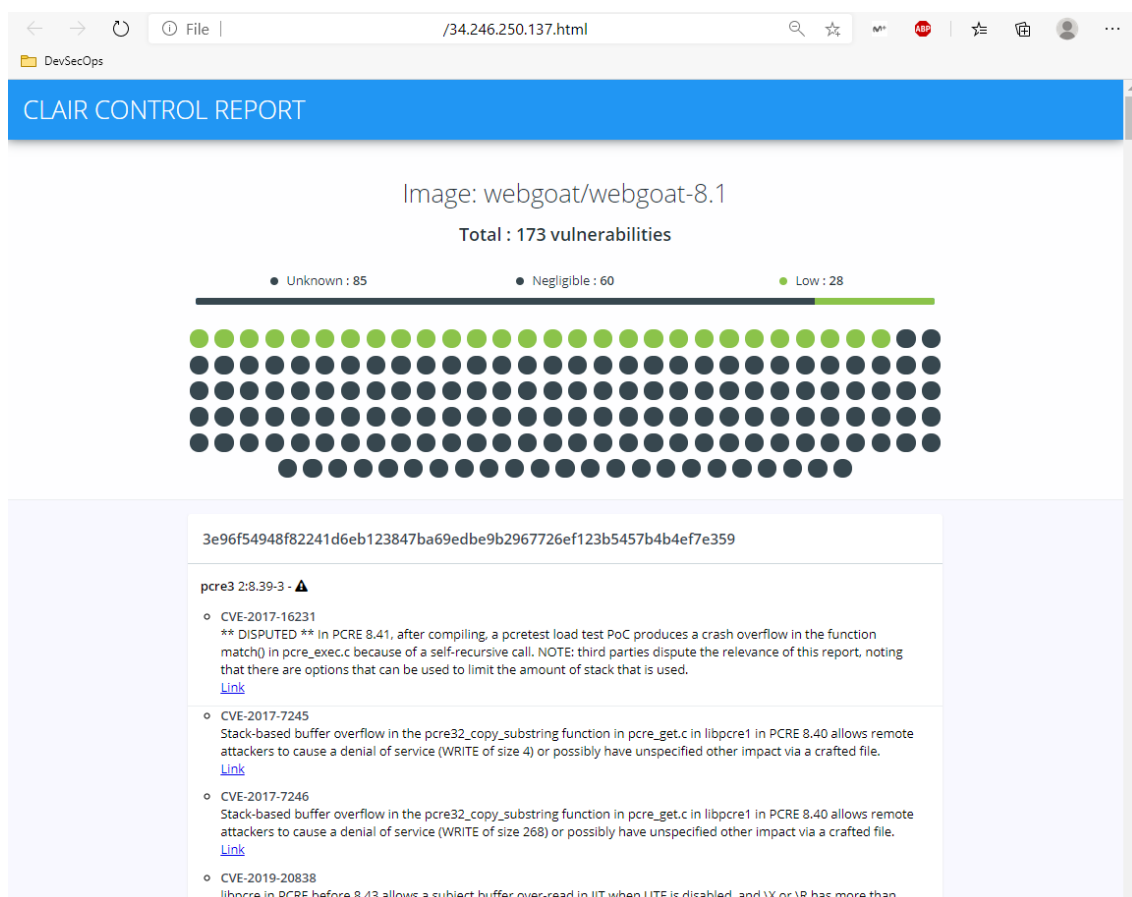


Ilustración 51 Reporte Clairctl como informe HTML

Tal y como puede apreciarse en las imágenes anteriores, los resultados de Clair determinan que es una imagen muy vulnerable, por lo que no debería ser subida a ningún repositorio para su despliegue.

5.1.7 Docker Deployment

En esta etapa se realiza el Despliegue Continuo (CD) de la aplicación localmente para poder realizar en la siguiente etapa el análisis DAST:

```
Jenkins > WebGoat_Pipeline > #65

[Pipeline] { (Docker Deployment)
[Pipeline] tool
[Pipeline] envVarsForTool
[Pipeline] withEnv
[Pipeline] {
[Pipeline] echo
Deploying the application
[Pipeline] sh
+ docker run -d --name webgoat-asc -p 8888:8888 --net 18-19_absace_myinet webgoat/webgoat-8.1
a957e3616651f6574edf429fe7ac01586e96da2d2f22c8e42fd6ad62084f98bb
[Pipeline] echo
http://webgoat-asc:8888/WebGoat
[Pipeline] sleep
Sleeping for 10 sec
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (DAST Analysis)
[Pipeline] tool
```

Ilustración 52 Éxito en etapa “Docker Deployment “

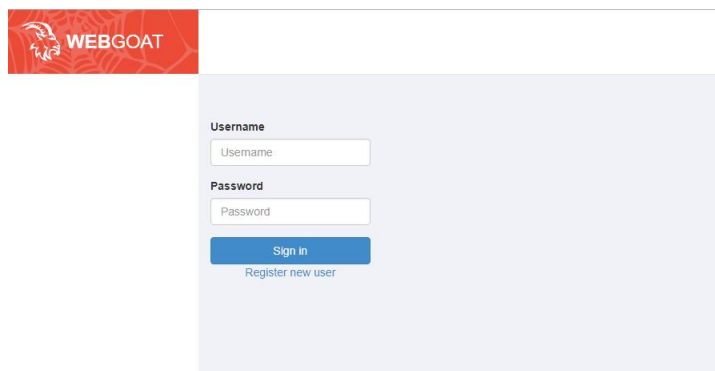


Ilustración 53 WebGoat desplegada para su análisis DAST

5.1.8 DAST Analysis

En el momento en el que la aplicación ha sido desplegada. es momento para la ejecución del análisis DAST con OWASP ZAP:

```
Jenkins > WebGoat_Pipeline > #65

[Pipeline] { (DAST Analysis)
[Pipeline] tool
[Pipeline] envVarsForTool
[Pipeline] withEnv
[Pipeline] {
[Pipeline] echo
DAST Analysis Starting
[Pipeline] sh
+ docker exec 18-19_absace_zap_1 zap-cli quick-scan --self-contained --start-options -config api.disablekey=true http://webgoat-asc:8888/WebGoat
[INFO] Starting ZAP daemon
[INFO] Running a quick scan for http://webgoat-asc:8888/WebGoat
[INFO] Issues found: 0
[INFO] Shutting down ZAP daemon
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Stop and Delete App Docker Container)
[Pipeline] tool
```

Ilustración 54 Éxito en etapa “DAST Analysis “

Con la configuración establecida, se observa que no se han detectado vulnerabilidades durante el despliegue de la aplicación.

5.1.9 Stop and Delete App Docker Container

Última etapa en la que se la detención del contenedor la aplicación web, así como la eliminación de este para la siguiente ejecución del Pipeline:

```
Jenkins > WebGoat_Pipeline > #65

[Pipeline] { (DAST Analysis)
[Pipeline] tool
[Pipeline] envVarsForTool
[Pipeline] withEnv
[Pipeline] {
[Pipeline] echo
DAST Analysis Starting
[Pipeline] sh
+ docker exec 18-19_obsace_zap_1 zap-cli quick-scan --self-contained --start-options -config api.disablekey=true http://webgoat-asc:8888/WebGoat
[INFO] Starting ZAP daemon
[INFO] Running a quick scan for http://webgoat-asc:8888/WebGoat
[INFO] Issues found: 0
[INFO] Shutting down ZAP daemon
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Stop and Delete App Docker Container)
[Pipeline] tool
[Pipeline] envVarsForTool
[Pipeline] withEnv
[Pipeline] {
[Pipeline] echo
Stopping and Deleting the App Docker Container
[Pipeline] sh
+ docker stop webgoat-asc
webgoat-asc
[Pipeline] sh
+ docker rm -f webgoat-asc
webgoat-asc
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Ilustración 55 Éxito en etapa “Stop and Delete App Docker Container”

Finalmente se puede observar que la ejecución global del Job es exitosa por completo, obteniéndose el resultado de la siguiente imagen:

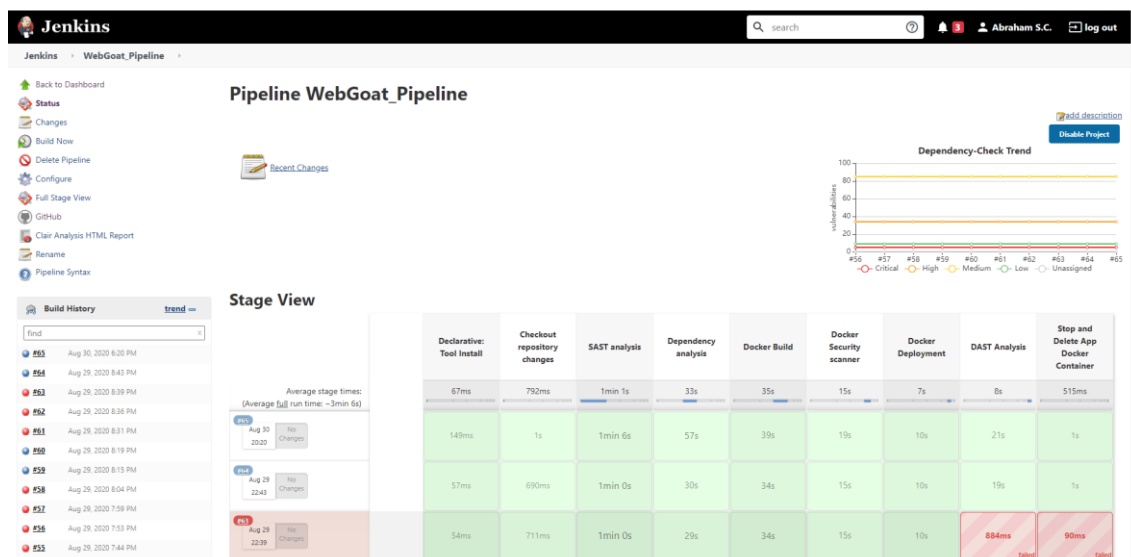


Ilustración 56 Visión global del Pipeline DevSecOps con ejecución exitosa

5.2 ENTORNO KUBERNETES

Tal y como se ha explicado en el apartado 4.4.3 Migración a Kubernetes con Kompose, se ha seguido la documentación oficial de la herramienta para la realización de la implementación y pruebas de esta.

Lamentablemente, no se ha podido alcanzar el objetivo de implementar la herramienta para poder desplegar el ecosistema DevSecOps en un entorno Kubernetes, tal y como puede apreciarse en la imagen siguiente:

```

PuTTY (inactive)
INFO Network mynet is detected at Source, shall be converted to equivalent NetworkPolicy at Destination
WARN Volume mount on the host "/home/ubuntu/18-19_absace/docker-compose-data/postgres-data" isn't supported - ignoring path on the host
INFO Network mynet is detected at Source, shall be converted to equivalent NetworkPolicy at Destination
INFO Network mynet is detected at Source, shall be converted to equivalent NetworkPolicy at Destination
INFO We are going to create Kubernetes Deployments, Services and PersistentVolumeClaims for your Dockerized application. If you need different kind of resources, use the 'kompose convert' and 'kubectl create -f' commands instead.

INFO Deploying application in "default" namespace
INFO Successfully created Service: clair
INFO Successfully created Service: jenkins
INFO Successfully created Service: sonarqube
INFO Successfully created Service: zap
INFO Successfully created PersistentVolumeClaim: clair-claim0 of size 100Mi. If your cluster has dynamic storage provisioning, you don't have to do anything. Otherwise you have to create PersistentVolume to make PVC work
INFO Successfully created PersistentVolumeClaim: clair-claim1 of size 100Mi. If your cluster has dynamic storage provisioning, you don't have to do anything. Otherwise you have to create PersistentVolume to make PVC work
INFO Successfully created PersistentVolumeClaim: clairctl-claim0 of size 100Mi. If your cluster has dynamic storage provisioning, you don't have to do anything. Otherwise you have to create PersistentVolume to make PVC work
INFO Successfully created PersistentVolumeClaim: clairctl-claim1 of size 100Mi. If your cluster has dynamic storage provisioning, you don't have to do anything. Otherwise you have to create PersistentVolume to make PVC work
INFO Successfully created PersistentVolumeClaim: jenkins-home of size 100Mi. If your cluster has dynamic storage provisioning, you don't have to do anything. Otherwise you have to create PersistentVolume to make PVC work
INFO Successfully created PersistentVolumeClaim: jenkins-claim1 of size 100Mi. If your cluster has dynamic storage provisioning, you don't have to do anything. Otherwise you have to create PersistentVolume to make PVC work
INFO Successfully created PersistentVolumeClaim: jenkins-claim2 of size 100Mi. If your cluster has dynamic storage provisioning, you don't have to do anything. Otherwise you have to create PersistentVolume to make PVC work
INFO Successfully created PersistentVolumeClaim: postgres-claim0 of size 100Mi. If your cluster has dynamic storage provisioning, you don't have to do anything. Otherwise you have to create PersistentVolume to make PVC work
INFO Successfully created PersistentVolumeClaim: zap of size 100Mi. If your cluster has dynamic storage provisioning, you don't have to do anything. Otherwise you have to create PersistentVolume to make PVC work

Your application has been deployed to Kubernetes. You can run 'kubectl get deployment,svc,pods,pvc' for details.
ubuntu@ip-172-31-41-31:~/18-19_absace$ minikube service jenkins
-----|-----|-----|-----|
| NAMESPACE | NAME   | TARGET PORT |    URL   |
-----|-----|-----|-----|
| default   | jenkins |             | No node port |
-----|-----|-----|-----|
* service default/jenkins has no node port
ubuntu@ip-172-31-41-31:~/18-19_absace$ kubectl describe svc jenkins
Name:                 jenkins
Namespace:            default
Labels:               io.kompose.service=jenkins
Annotations:          kompose.cmd: kompose up --server https://172.17.0.3:6443
                     kompose.version: 1.21.0 (992df58d8)
Selector:             io.kompose.service=jenkins
Type:                 ClusterIP
IP:                   10.104.180.204
Port:                 8080
TargetPort:           8080/TCP
Endpoints:            <none>
Port:                 50000
TargetPort:           50000/TCP
Endpoints:            <none>
Session Affinity:     None
Events:               <none>
ubuntu@ip-172-31-41-31:~/18-19_absace$

```

Ilustración 57 Implementación y pruebas de Kompose

La dedicación de tiempo y recursos ha sido mucho menor de la esperada debido a las Dificultades encontradas durante la realización de este Trabajo.

6 DIFICULTADES ENCONTRADAS

La elaboración de este Trabajo ha generado dificultades que se han ido presentando durante todo el proceso de investigación, codificación, implementación, y pruebas de este, a medida que la complejidad del Trabajo se iba incrementando.

6.1 DOCKER

Al hacer uso de la imagen oficial de Jenkins se dio por supuesto que no habría problemas para poder integrar Docker en Jenkins. Era necesaria su utilización para poder compilar la imagen de la aplicación web. Fue imposible integrarla con la configuración que permite el servidor una vez levantando, ni con instalación dentro del mismo. No tenía permisos para poder hacer uso del *Docker Daemon* de la máquina Host a través del *sock* de Docker aun especificándolo en el esquema YAML de *Docker-Compose*.

Esta dificultad fue solventada al crear una imagen Docker propia de Jenkins haciendo uso de un fichero Dockerfile para la instalación de Docker nativamente, dando así permisos para poder hacer uso del *Docker Daemon* de la máquina Host a través del *sock* de Docker.

6.2 WEBGOAT

Como ya se ha explicado en el punto 3.6.1 WebGoat, se ha utilizado el código publicado del proyecto en el repositorio Github [54]. Durante un tiempo demasiado prolongado se ha producido un error durante la etapa de “Creación de Binarios e Imágenes Docker”, debido a un error de codificación en el fichero Dockerfile para la creación de la imagen Docker a partir de los binarios que se generan usando Maven [71].

Se dio por supuesto que ese código no tenía errores de codificación en ese aspecto, por lo que se tuvo que invertir mucho más tiempo del esperado para dar con el inconveniente y solventarlo ya que se pensaba en un primer momento que era un problema de una integración inadecuada de la herramienta Docker en Jenkins, y que no se hacía un uso adecuado por parte del servidor.

Finalmente, haciendo un análisis detallado de los resultados de las ejecuciones del Job, tanto de los binarios en el Workspace como de todos los logs generados por el servidor, se detectó el problema y se solventó correctamente.

6.3 CLAIR Y CLAIRCTL

Clair necesita un fichero de configuración YAML para poder configurar todo su ecosistema: servicio de análisis, base de datos, servicios expuestos para la integración con clientes terceros, servicio de actualización de fuentes de vulnerabilidades.

Los inconvenientes encontrados fueron la duplicidad en la publicación de los puertos de Clair, tanto en el fichero de configuración como en el YAML de *Docker-Compose*, puesto que era necesario hacer esta publicación en ambas partes.

Durante un tiempo demasiado prolongado no se estaba publicando los puertos en el YAML de *Docker-Compose*, por lo que no se integraba el servicio de Clair en el ecosistema, dando así fallo catastrófico. Se solventó creando una nueva instancia y desplegando Clair en solitario, hasta que se descubrió el problema que lo provocaba.

6.4 HERRAMIENTAS Y DESPLIEGUES NO REALIZADOS

Debido a la inversión de tiempo dedicada a solventar las dificultades descritas anteriormente, algunos de los objetivos planteados no han podido realizarse, dado que este Trabajo se lleva a cabo en un tiempo limitado.

Por esta razón no ha podido realizarse la implementación de la herramienta Notary, así como la implementación y migración de todo el sistema a Kubernetes haciendo uso de Kompose.

6.4.1 Kompose

En el caso de esta herramienta, se han encontrado dificultades con la configuración y el despliegue del sistema Kubernetes. También se han encontrado problemas a la hora de hacer una conversión del esquema YAML de *Docker-Compose* ya que actualmente Kompose no soporta versiones posteriores a la versión 3 de *Docker-Compose*.

A continuación, se encuentran algunas de las dificultades encontradas durante la implementación de Kubernetes usando esta herramienta:

- <https://discuss.kubernetes.io/t/the-connection-to-the-server-host-6443-was-refused-did-you-specify-the-right-host-or-port/552>
- <https://github.com/kubernetes/minikube/issues/7663>

Junto con la inmensa inversión de tiempo para resolver las dificultades anteriores, ha sido muy complicado poder dedicarle el tiempo necesario a esta herramienta para hacer un uso exitoso de la misma en este Trabajo.

6.4.2 Notary

En el caso de esta herramienta, y debido a la limitación de tiempo para el desarrollo de este Trabajo, finalmente no se ha podido realizar ni el estudio ni la implementación que se había marcado en los objetivos de este Trabajo.

7 CONCLUSIONES

En la actualidad la seguridad normalmente no está contemplada en el ciclo de vida de un proyecto software, es más, se suele dejar para añadir al final de dicho ciclo haciendo así que solo sea un mero parche, provocando que las deficiencias en seguridad sean acusadas, sobre todo en el futuro. A fin de cuentas, para que un software sea seguro debe cumplir los tres pilares de la seguridad: integridad, confidencialidad, y disponibilidad.

El objetivo principal de este Trabajo era el diseño e implementación de un ciclo de vida del desarrollo software seguro, implementando de un Sistema *Continuous Integration (CI) / Continuous Deployment (CD)*, es decir, CI/CD, en un entorno Docker, al cual se le integrarían diferentes herramientas de seguridad durante el flujo del Pipeline para convertirlo en DevSecOps.

Crear un Pipeline sencillo en un servidor CI/CD (Jenkins) no es complejo y nos será de gran ayuda a la hora de mantener y monitorizar el flujo de vida de nuestro proyecto software. En el momento en el que se quiere añadir seguridad en todas las etapas al Pipeline para convertirlo en DevSecOps, salen a la luz una serie de problemas e inconvenientes tediosos que hacen resaltar la complejidad del objetivo a alcanzar.

Con este Trabajo queda de manifiesto que la capacidad de recursos del Host donde se despliegue este sistema es limitante a la hora de poder hacer uso completa y perfectamente del mismo, puesto que los inconvenientes encontrados en el anterior TFM (1.1 Tema del Trabajo) han sido solventados en este Trabajo.

La mayor complejidad radica en poder integrar dichas herramientas en Jenkins, concretamente en su flujo de Pipeline, siendo necesario tener perfectamente configurados los plugins necesarios para la armonía del sistema. En el momento en el que esto suceda, el desarrollo de un proyecto software con un Pipeline DevSecOps será mucho más sencilla que con un simple DevOps.

Para poder conseguirlo, se debe modularizar el problema dividiéndolo en partes más pequeñas, lo que provoca que se divida el espacio de estados del sistema original, puesto que cada módulo tiene una limitación de estados.

La razón de lo anterior es sencilla: si se realizan cambios pequeños, y en cada uno de ellos se realiza la integración continua, es mucho más fácil encontrar el fallo en un bloque de código acotado, mientras que, si hacemos modificaciones grandes y las integramos de una sola vez, sería demasiado complicado de encontrar.

Se ha demostrado en este Trabajo la importancia de añadir la seguridad en todas las etapas, resaltando la idea de que seguridad y desarrollo son compatibles, alejando la práctica de implementar la seguridad al final del ciclo de vida del desarrollo software:

- Se ha diseñado un ciclo de vida del desarrollo software seguro, definiendo las diferentes etapas que lo componen.
- Se ha implementado un sistema CI/CD basado en servidor para la implementación de un Pipeline DevSecOps

- Se ha llevado a cabo el desarrollo de un pipeline, que incluye todas las etapas que deben pasar un proyecto software. Este proceso sigue la metodología de desarrollo software, por lo que los cambios en el repositorio que no hayan sido tratados en cada etapa no serán promocionados.
- Se ha realizado de manera automatizada, sin requerir de la interacción manual de ningún miembro para realizar el checkout del código, compilar y desplegar, así como la ejecución de todas las pruebas de seguridad.

En consecuencia, implementar un sistema CI/CD teniendo en cuenta los requisitos y las políticas de seguridad, así como la implementación de seguridad SAST, DAST, análisis de dependencias, análisis de imágenes Docker, nos proporcionan un nivel de información muy elevado sobre la calidad del producto software que se está desplegando, así como los riesgos de seguridad que tiene su entrega al mercado.

8 LÍNEAS DE TRABAJOS FUTUROS

Tras finalizar este Trabajo, y teniendo en cuenta los resultados obtenidos, se proponen las siguientes cuestiones para un trabajo futuro:

- Realizar un estudio y la integración adecuada de la herramienta Notary, la cual se utiliza para analizar y asegurar que las imágenes Docker que se están utilizando han sido firmadas, comprobando así su integridad, confidencialidad, y disponibilidad.
- Realizar un estudio e implementación más completo del realizado en este trabajo para el análisis DAST utilizando la herramienta OWASP ZAP, ya que no se han conseguido resultados lo suficientemente claros en este Trabajo.
- Realizar un estudio y mejora continua del script del Pipeline, ya siempre es posible encontrar mejoras en el rendimiento y funcionalidad de este, siguiendo la metodología DevSecOps.
- Realizar un estudio necesario para la migración y adaptación de todo el ecosistema DevSecOps creado en este Trabajo a un entorno Kubernetes, analizando la viabilidad de hacer uso de la herramienta Kompose u otras herramientas.

9 BIBLIOGRAFÍA Y REFERENCIAS

- [1] GSyC, «DevOps,» URJC, [En línea]. Available: <https://gsyc.urjc.es/~mortuno/lagrs/06-DevOps.pdf>.
- [2] RedHat, «What is DevSecOps?,» 2019. [En línea]. Available: <https://www.redhat.com/en/topics/devops/what-is-devsecops>.
- [3] CodeURJC, «URJC Forge,» [En línea]. Available: <https://github.com/codeurjc/codeurjc-forge>.
- [4] A. Santana, «TFM DevSecOps Demo Pipeline Abraham Santana MCyP URJC,» [En línea]. Available: <https://youtu.be/HkFP6tAobp0>.
- [5] Amazon, «Que es Amazon Web Services,» [En línea]. Available: <https://aws.amazon.com/es/what-is-aws/>.
- [6] Wikipedia, «Volatility, Uncertainty, Complexity, and Ambiguity,» [En línea]. Available: https://en.wikipedia.org/wiki/Volatility,_uncertainty,_complexity_and_ambiguity.
- [7] Wikipedia, «Metodología de desarrollo de software,» [En línea]. Available: https://es.wikipedia.org/wiki/Metodolog%C3%ADa_de_desarrollo_de_software.
- [8] Centers for Medicare & Medicaid Services, «Selecting a Development Approach,» 27 marzo 2008. [En línea]. Available: <https://web.archive.org/web/20190102182947/https://www.cms.gov/Research-Statistics-Data-and-Systems/CMS-Information-Technology/XLC/Downloads/SelectingDevelopmentApproach.pdf>.
- [9] Wikipedia, «Modelo cascada realimentado para el ciclo de vida.,» [En línea]. Available: https://es.wikipedia.org/wiki/Software#Modelo_cascada.
- [10] Wipikedia, «Modelo iterativo incremental,» [En línea]. Available: https://es.wikipedia.org/wiki/Software#Modelo_iterativo_incremental.
- [11] J. MUKHERJEE, «Atlassian,» 20 06 2020. [En línea]. Available: <https://www.atlassian.com/es/continuous-delivery/principles/devsecops>.
- [12] VERACODE, «Estado de Seguridad del Software,» 2019. [En línea]. Available: <https://www.veracode.com/state-of-software-security-report>.
- [13] N. W. Edmund, «The General Pattern of the Scientific Method (SM-14),» ISBN-0-9632866-3-3, 1994. [En línea]. Available: <https://eric.ed.gov/?id=ED393871>.
- [14] Docker, «Docker Documentation,» [En línea]. Available: <https://www.docker.com/resources/what-container>.
- [15] MCyP, Software Seguro. Virtualización y Contenedores Docker, Módulo 4 (Unidad 4.3).
- [16] Docker, «Docker architecture,» [En línea]. Available: <https://docs.docker.com/get-started/overview/#docker-architecture>.

- [17] M. Fowler, «Continuous Integration,» [En línea]. Available: <https://martinfowler.com/articles/continuousIntegration.html>. [Último acceso: 4 6 2019].
- [18] Apache, «Maven,» [En línea]. Available: <https://maven.apache.org/>.
- [19] Jenkins, «Jenkins Documentation,» [En línea]. Available: <https://www.jenkins.io/doc/>.
- [20] Wikipedia, «Jenkins,» [En línea]. Available: <https://es.wikipedia.org/wiki/Jenkins>. [Último acceso: 8 6 2019].
- [21] Jenkins, «Jenkins as Docker,» [En línea]. Available: <https://hub.docker.com/r/jenkins/jenkins/>.
- [22] A. Santana, «jenkins-dockerfile,» [En línea]. Available: https://github.com/MCYP-UniversidadReyJuanCarlos/18-19_absace/blob/master/jenkins-dockerfile.
- [23] Wikipedia, «Static program analysis,» [En línea]. Available: https://en.wikipedia.org/wiki/Static_program_analysis. [Último acceso: 22 05 2019].
- [24] Wikipedia, «List of tools for static code analysis,» [En línea]. Available: https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis. [Último acceso: 22 05 2019].
- [25] Wikipedia, «Linters,» [En línea]. Available: [https://en.wikipedia.org/wiki/Lint_\(software\)](https://en.wikipedia.org/wiki/Lint_(software)). [Último acceso: 20 07 2020].
- [26] Wikipedia, «Code Smell,» [En línea]. Available: https://en.wikipedia.org/wiki/Code_smell. [Último acceso: 06 07 2020].
- [27] Wikipedia, «Cyclomatic Complexity,» [En línea]. Available: https://en.wikipedia.org/wiki/Cyclomatic_complexity. [Último acceso: 06 07 2020].
- [28] The MITRE Corporation, «Common Vulnerabilities and Exposures,» [En línea]. Available: <https://cve.mitre.org/>.
- [29] Synopsys, «Enhanced Vulnerability Data,» [En línea]. Available: <https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis/vulnerability-reporting.html>. [Último acceso: 06 07 2020].
- [30] SonarQube, «Issues Doc.,» [En línea]. Available: <https://docs.sonarqube.org/latest/user-guide/issues/>. [Último acceso: 06 07 2020].
- [31] sonarqube, «Issues on sonarqube,» 2019. [En línea]. Available: <https://docs.sonarqube.org/latest/user-guide/issues/>. [Último acceso: 23 05 2019].
- [32] Eclipse, «Eclipse Foundation,» [En línea]. Available: <https://www.eclipse.org/>. [Último acceso: 06 07 2020].
- [33] Visual Studio Code, «Web Site,» [En línea]. Available: <https://code.visualstudio.com/>. [Último acceso: 06 07 2020].

- [34] Sublime Text, «Web Site,» [En línea]. Available: <https://www.sublimetext.com/>. [Último acceso: 06 07 2020].
- [35] OWASP, «Source Code Analysis Tools,» [En línea]. Available: https://owasp.org/www-community/Source_Code_Analysis_Tools. [Último acceso: 06 07 2020].
- [36] SonarSource, «Web Site,» [En línea]. Available: <https://www.sonarsource.com/>. [Último acceso: 06 07 2020].
- [37] SonarQube, «Web Site,» [En línea]. Available: <https://www.sonarqube.org/>. [Último acceso: 06 07 2020].
- [38] SonarSource, «SonarQube as Docker,» [En línea]. Available: https://hub.docker.com/_/sonarqube.
- [39] SonarQube, «SonarQube Scanner (plugin Jenkins),» [En línea]. Available: <https://plugins.jenkins.io/sonar/>.
- [40] OWASP, «OWASP Dependency-Check,» jeremylong, octubre 2012. [En línea]. Available: <https://owasp.org/www-project-dependency-check/>.
- [41] OWASP, «OWASP Dependency-Check Documentation,» jeremylong, 2012. [En línea]. Available: <https://jeremylong.github.io/DependencyCheck/>.
- [42] OWASP, «OWASP Dependency-Check Jenkins Plugin,» Steve Springett, agosto 2013. [En línea]. Available: <https://plugins.jenkins.io/dependency-check-jenkins-plugin/>.
- [43] CoreOS, «Clair,» Clair, [En línea]. Available: <https://coreos.com/clair/docs/latest/>.
- [44] CoreOS, «Integration Tools Documentation,» Clair, 20 marzo 2019. [En línea]. Available: <https://github.com/quay/clair/blob/master/Documentation/integrations.md>.
- [45] CoreOS, «Clair as Docker,» [En línea]. Available: <https://quay.io/repository/coreos/clair?tab=tags>.
- [46] CoreOS, «Clair project,» [En línea]. Available: <https://github.com/quay/clair>.
- [47] J. Garcia Gonzalez, «Clairctl,» jgsquare, 7 julio 2018. [En línea]. Available: <https://github.com/jgsquare/clairctl>.
- [48] A. Coralic, «Clair-Scanner,» arminc, 6 julio 2019. [En línea]. Available: <https://github.com/arminc/clair-scanner>.
- [49] The Web Application Security Consortium, «Web Application Security Scanner Evaluation Criteria,» febrero 2014. [En línea]. Available: <http://projects.webappsec.org/w/page/13246986/Web%20Application%20Security%20Scanner%20Evaluation%20Criteria>.
- [50] NIST, «Software Assurance Tools: Web Application Security ScannerFunctional Specification Version 1.0,» Web Application Scanner, 2008. [En línea]. Available: https://samate.nist.gov/docs/webapp_scanner_spec_sp500-269.pdf.

- [51] OWASP, «Zed Attack Proxy (ZAP),» Simon Bennetts (psiinon), Ricardo Pereira (thc202), Rick Mitchell (kingthorin), [En línea]. Available: <https://owasp.org/www-project-zap/>.
- [52] OWASP, «OWASP ZAP as Docker,» [En línea]. Available: <https://hub.docker.com/r/owasp/zap2docker-stable/>.
- [53] <https://owasp.org/>, «OWASP Foundation,» 2004. [En línea]. Available: <https://owasp.org/>.
- [54] OWASP, «OWASP WebGoat,» 2 febrero 2016. [En línea]. Available: <https://owasp.org/www-project-webgoat/>.
- [55] Kubernetes, «Kubernetes Documentation,» [En línea]. Available: <https://kubernetes.io/es/docs/home/>.
- [56] Kubernetes, «Kubernetes GitHub Project,» [En línea]. Available: <https://github.com/kubernetes/kubernetes>.
- [57] Cloud Native Computing Foundation, «Cloud Native Computing Foundation,» [En línea]. Available: <https://www.cncf.io/>.
- [58] Kompose, «Kompose = Kubernetes + Compose,» [En línea]. Available: <https://kompose.io/>.
- [59] Kompose, «Kompose (Kubernetes + Compose) GitHub,» [En línea]. Available: <https://github.com/kubernetes/kompose>.
- [60] Kubernetes, «kubectl install and config,» [En línea]. Available: <https://kubernetes.io/es/docs/tasks/tools/install-kubectl/>.
- [61] minikube, «minikube GitHub,» [En línea]. Available: <https://github.com/kubernetes/minikube>.
- [62] Kubernetes, «minikube install and config,» [En línea]. Available: <https://kubernetes.io/es/docs/tasks/tools/install-minikube/>.
- [63] Cybersecurity Security and Infrastructure Security Agency (CISA), «Secure Software Development Life Cycle Processes,» Noopur Davis, [En línea]. Available: <https://us-cert.cisa.gov/bsi/articles/knowledge/sdlc-process/secure-software-development-life-cycle-processes>.
- [64] M. Krief, Learning DevOps, Packt Publishing, 2019.
- [65] Putty, «PuTTY is an SSH and telnet client,» [En línea]. Available: <https://www.putty.org/>.
- [66] MCyP URJC, «Repositorio del TFM del MCyP de Abraham en la URJC,» Abraham Santana Cebrián, septiembre 2020. [En línea]. Available: https://github.com/MCYP-UniversidadReyJuanCarlos/18-19_absace.
- [67] Jenkins, «Installing Jenkins,» [En línea]. Available: <https://www.jenkins.io/doc/book/installing/>.
- [68] Jenkins, «Repositorio de plugins de Jenkins,» [En línea]. Available: <https://plugins.jenkins.io/>.

- [69] SonarSource, «Try Out SonarQube,» SonarQube, [En línea]. Available: <https://docs.sonarqube.org/latest/setup/get-started-2-minutes/>.
- [70] The MITRE Corporation, «Common Weakness Enumeration,» [En línea]. Available: <https://cwe.mitre.org/>.
- [71] A. S. Cebrian, «Subsanación Error de codificación fichero Dockerfile WebGoat 8.1,» [En línea]. Available: <https://github.com/Abrin09/WebGoat/commit/c229c56c3b2f9c8f9633963dc54b848b08177fd2>.
- [72] MCyP, Software Seguro. Integración Continua y Despliegue Continuo., Módulo 4 (Unidad 4.5).
- [73] Docker, «Introduction to container security,» Agosto 2016. [En línea]. Available: <https://goto.docker.com/intro-container-security.html>.
- [74] Miju Han, GitHub, «Introducing security alerts on GitHub,» [En línea]. Available: <https://github.blog/2017-11-16-introducing-security-alerts-on-github/>.
- [75] Docker, «Intro to Container Security,» Marzo 2015. [En línea]. Available: https://d3oypxn00j2a10.cloudfront.net/assets/img/Docker%20Security/WP_Intro_to_container_security_03.20.2015.pdf.
- [76] Wikipedia, «Dynamic program analysis,» [En línea]. Available: https://en.wikipedia.org/wiki/Dynamic_program_analysis. [Último acceso: 22 05 2019].
- [77] J. Leonard, «DevSecOps - combining speed and security,» Agosto 2016. [En línea]. Available: <https://search.proquest.com/docview/1810884740>.
- [78] Wikipedia, «DevOps,» 6 diciembre 2018. [En línea]. Available: <https://es.wikipedia.org/wiki/DevOps>.
- [79] OWASP, «Configuración servicio ZAP,» ZAP, [En línea]. Available: <https://www.zaproxy.org/getting-started/>.
- [80] Instituto de Formación Profesional - Libros Digitales, «Ciclo de vida del Software y Modelos de desarrollo,» 15 febrero 2010. [En línea]. Available: https://web.archive.org/web/20100215155237/http://www.cepeu.edu.py/LIBROS_ELECTRONIC_OS_3/lpcu097%20-%2001.pdf.
- [81] E. Viitasuo, «Adding security testing in DevOps software development with continuous integration and continuous delivery practices,» Bachelor's thesis, May 2020. [En línea]. Available: <https://www.theseus.fi/handle/10024/342349>.
- [82] R. Chandramouli, «Security Assurance Requirements for Linux Application Container Deployments,» NIST, Octubre 2017. [En línea]. Available: <https://csrc.nist.gov/publications/detail/nistir/8176/final>.

10 TABLA DE ILUSTRACIONES

Ilustración 1 Modelo cascada realimentado para el ciclo de vida [9].....	6
Ilustración 2 Modelo iterativo incremental para el ciclo de vida [80]	6
Ilustración 3 Tareas planificadas	11
Ilustración 4 Planificación con el Diagrama de Gantt	11
Ilustración 5. Esquema conceptual del funcionamiento de Docker [15] [16].....	12
Ilustración 6 Esquema conceptual de Integración Continua y Despliegue Continuo [72]	13
Ilustración 7. Visión general de un Pipeline en Jenkins	14
Ilustración 8. Esquema de detección de vulnerabilidades por BlackDuck [29]	17
Ilustración 9 Esquema del framework de Clair	22
Ilustración 10 Reporte HTML del análisis Clair generado por Clairctl.....	23
Ilustración 11 Reporte por consola del análisis Clair generado por Clair-Scanner	24
Ilustración 12 Configuración servicio OWASP ZAP” man-in-the-middle” [79].....	25
Ilustración 13 Aplicación WebGoat desplegada y funcionando.....	26
Ilustración 14 Etapas generales en DevOps	28
Ilustración 15 Etapas DevSecOps en este Trabajo	28
Ilustración 16. Configuración Instancia Ubuntu usada en AWS.....	32
Ilustración 17 Despliegue ecosistema DevSecOps usando Docker-Compose vía Putty (1)	33
Ilustración 18 Despliegue ecosistema DevSecOps usando Docker-Compose vía Putty (2)	33
Ilustración 19 Despliegue ecosistema DevSecOps usando Docker-Compose vía Putty (3)	34
Ilustración 20 Despliegue ecosistema DevSecOps usando Docker-Compose vía Putty (4)	34
Ilustración 21 Despliegue ecosistema DevSecOps usando Docker-Compose vía Putty (5)	34
Ilustración 22. Configuración de la sección Maven Project	37
Ilustración 23. Configuración del servidor SonarQube	38
Ilustración 24 Configuración del servidor OWASP ZAP	38
Ilustración 25. Configuración de Maven en la configuración global de Jenkins	39
Ilustración 26 Configuración de JDK en la configuración global de Jenkins.....	40
Ilustración 27. Configuración de SonarQube Scanner en la configuración global de Jenkins	40
Ilustración 28. Configuración de Git en la configuración global de Jenkins.....	41

Ilustración 29. Servidor SonarQube recién configurado esperando analizar	41
Ilustración 30. Creación del Token para SonarQube.....	42
Ilustración 31. Etapas configuradas del Pipeline DevSecOps en Jenkins	43
Ilustración 32. Visión global del Pipeline DevSecOps configurado y funcionando	43
Ilustración 33. Opciones de configuración del Pipeline DevSecOps (1).....	44
Ilustración 34. Opciones de configuración del Pipeline DevSecOps (2).....	44
Ilustración 35 Configuración y despliegue de sistema Kubernetes con Kompose.....	48
Ilustración 36 Éxito en etapa “Declarative: Tool Install “	49
Ilustración 37 Éxito en etapa “Checkout repository changes “	49
Ilustración 38 Éxito en etapa “SAST analysis (1)”	50
Ilustración 39 Éxito en etapa “SAST analysis (1) “	50
Ilustración 40 Proyecto WebGoat en SonarQube	51
Ilustración 41 Información general del análisis sobre el proyecto WebGoat	51
Ilustración 42 Vulnerabilidades bloqueantes en el proyecto WebGoat	52
Ilustración 43 Detalle de una vulnerabilidad bloqueante del proyecto WebGoat	52
Ilustración 44 Éxito en etapa “Dependency analysis “	53
Ilustración 45 Resultados de OWASP Dependency Check (1)	53
Ilustración 46 Resultados de OWASP Dependency Check (2)	54
Ilustración 47 Éxito creando binarios en etapa “Release and Docker Build “	54
Ilustración 48 Éxito creando imagen Docker en etapa “Release and Docker Build “	55
Ilustración 49 Éxito utilizando Clairctl en etapa “Docker security scanner “	55
Ilustración 50 Éxito utilizando Clair-Scanner en etapa “Docker security scanner “	56
Ilustración 51 Reporte Clairctl como informe HTML.....	56
Ilustración 52 Éxito en etapa “Docker Deployment “	57
Ilustración 53 WebGoat desplegada para su análisis DAST	57
Ilustración 54 Éxito en etapa “DAST Analysis “	57
Ilustración 55 Éxito en etapa “Stop and Delete App Docker Container “	58
Ilustración 56 Visión global del Pipeline DevSecOps con ejecución exitosa	58
Ilustración 57 Implementación y pruebas de Kompose.....	59

11 ANEXO 1: DOCKER COMPOSE FILE SCRIPT

Contenido del fichero Dockerfile para la orquestación del ecosistema DevSecOps objeto de este Trabajo de Fin de Máster:

```
version: "3.8"
services:
  jenkins:
    build:
      dockerfile: jenkins-dockerfile
      context: .
    ports:
      - 8080:8080
      - 50000:50000
    networks:
      - mynet
    volumes:
      - jenkins_home:/var/jenkins_home
      - /var/run/docker.sock:/var/run/docker.sock
      - /usr/local/bin/docker:/usr/local/bin/docker
  sonarqube:
    image: sonarqube:lts
    ports:
      - 9000:9000
    networks:
      - mynet
  postgres:
    image: postgres:9.6
    restart: unless-stopped
    volumes:
      - ./docker-compose-data/postgres-data:/var/lib/postgresql/data:rw
    environment:
      - POSTGRES_PASSWORD=ChangeMe
      - POSTGRES_USER=clair
      - POSTGRES_DB=clair
    networks:
      - mynet
  clair:
    image: quay.io/coreos/clair:v2.0.6
    command: -config=/config/config.yaml
    #command: -log-level=debug
    ports:
      - 6060:6060
      - 6061:6061
    depends_on:
      - postgres
    volumes:
      - ./docker-compose-data/clair-config:/config:ro
      - ./docker-compose-data/clair-tmp:/tmp:rw
    user: root
    networks:
      - mynet
  clairctl:
    image: jgsquare/clairctl:latest
    restart: unless-stopped
    environment:
      - DOCKER_API_VERSION=1.40
    volumes:
      - ./docker-compose-data/clairctl-reports:/reports
      - /var/run/docker.sock:/var/run/docker.sock
    depends_on:
      - clair
    networks:
      - mynet
    user: root
  zap:
    image: owasp/zap2docker-stable
    restart: unless-stopped
    ports:
      - 8000:8000
    entrypoint: zap-x.sh -daemon -host 0.0.0.0 -port 8000 -config api.addrs.addr.name=.* -config
api.addrs.addr.regex=true -config api.key=5364864132243598723485
    volumes:
      - zap:/zap/data
networks:
  mynet:
    driver: bridge
volumes:
  jenkins_home:
  clair-postgres:
  zap:
```


12 ANEXO 2: PIPELINE SCRIPT

Contenido del script Pipeline DevSecOps implementado en un Job de tipo Pipeline en Jenkins:

```

pipeline {
    agent any
    environment {
        //Variables de entorno
        sonarURL="http://sonarqube:9000"
    }
    tools {
        maven 'maven363'
    }
    stages {
        stage('Checkout repository changes') {
            steps {
                echo 'Downloading the code from GitHub'
                checkout([$class: 'GitSCM', branches: [[name: '*/develop']],
                    doGenerateSubmoduleConfigurations: false, extensions: [],
                    submoduleCfg: [], userRemoteConfigs: [[url:
'https://github.com/Abrin09/WebGoat.git']]])
            }
        }
        stage('SAST analysis') {
            steps {
                echo 'Performing SAST analysis to the code from the commit'
                sh "/var/jenkins_home/tools/hudson.plugins.sonar.SonarRunnerInstallation/SonarQubeScanner/bin/sonar-scanner -X -Dsonar.host.url=$sonarURL -Dsonar.project-
Key=webgoat -Dsonar.projectName=webgoat -Dsonar.projectVersion=1.0 -Dsonar.java.bina-
ries=. -Dsonar.exclusions=**/*.ts"
                echo 'SAST Finished!!!'
            }
        }
        stage('Dependency analysis') {
            steps {
                echo 'Performing Code Dependency analysis from the commit'
                dependencyCheck additionalArguments: '', odcInstallation: 'dependen-
cyCheckOWASP'
                echo 'Finished!!!'
                dependencyCheckPublisher pattern: 'dependency-check-report.xml'
                echo 'Publishing results...'
            }
        }
        stage('Release and Docker Build') {
            steps {
                echo 'Compiling Code'
                sh "mvn clean install -DskipTests"
                echo 'Compiled Code Successful'
                echo 'Building docker image'
                sh "docker build -t webgoat/webgoat-8.1 ./webgoat-server"
                echo 'Docker image builded'
            }
        }
        stage ('Docker Security scanner') {
            steps {
                echo 'To Do integration of Clair with Pipeline'
                sh "docker exec -it 18-19_absace_clairctl_1 ls -la /reports/html || exit
0"
                sh "docker exec 18-19_absace_clairctl_1 clairctl analyze -l
webgoat/webgoat-8.1 || exit 0"
                sh "docker exec 18-19_absace_clairctl_1 clairctl report -l
webgoat/webgoat-8.1 || exit 0"
                sh "mkdir -p clairctl-reports"
                sh "docker cp 18-19_absace_clairctl_1:/reports/html/analysis-webgoat-
webgoat-8.1-latest.html ${WORKSPACE}/clairctl-reports/analysis-webgoat-8.1-
latest.html"
                sh '''
                DOCKER_GATEWAY=$(ip r | tail -n1 | awk '{ print $9 }')
                wget -qO clair-scanner https://github.com/arminc/clair-scanner/re-
leases/download/v12/clair-scanner_linux_amd64 && chmod +x clair-scanner
                ./clair-scanner --ip="$DOCKER_GATEWAY" --clair=http://clair:6060
webgoat/webgoat-8.1 || exit 0
                '''
                echo 'Publishing the HTML Report'
                sh "mkdir -p Clair-Analysis-Report"
            }
        }
    }
}

```

```

        publishHTML([allowMissing: false, alwaysLinkToLastBuild: false, keepAll:
false, reportDir: 'clairctl-reports',
                    reportFiles: 'analysis-webgoat-webgoat-8.1-latest.html', re-
portName: 'Clair Analysis HTML Report', reportTitles: 'Clair Analysis Report'])
        echo 'Publishing succesfully done'
    }
}
stage('Docker Deployment') {
    steps {
        echo 'Deploying the application'
        sh "docker run -d --name webgoat-asc -p 8888:8080 --net 18-
19_absace_mynet webgoat/webgoat-8.1 || exit 0"
        echo 'http://webgoat-asc:8888/WebGoat'
        sleep 10
    }
}
stage('DAST Analysis') {
    steps {
        echo 'DAST Analysis Starting'
        sh "docker exec 18-19_absace_zap_1 zap-cli quick-scan --self-contained -
-start-options '-config api.disablekey=true' http://webgoat-asc:8888/WebGoat"
    }
}
stage('Stop and Delete App Docker Container') {
    steps {
        echo 'Stoping and Deleting the App Docker Container'
        sh "docker stop webgoat-asc || exit 0"
        sh "docker rm -f webgoat-asc || exit 0"
    }
}
}
}
}

```