uoc.edu

DevSecOps: integración de herramientas SAST, DAST y de análisis de Dockers en un sistema de integración continua.

José Joaquín Caño Quintero
Máster Seguridad de las TIC
Seguridad empresarial

Pau del Canto Rodrigo
Víctor García Font

04/06/2019

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-SinObraDerivada 3.0 España de Creative Commons

FICHA DEL TRABAJO FINAL

Título del trabajo:	*DevSecOps: integración de herramientas SAST, DAST y de análisis de Dockers en un sistema de integración continua.*
Nombre del autor:	*José Joaquín Caño Quintero*
Nombre del consultor/a:	*Pau del Canto Rodrigo*
Nombre del PRA:	*Víctor García Font*
Fecha de entrega (mm/aaaa):	06/2019
Titulación::	*Máster Seguridad de las TIC*
Área del Trabajo Final:	*Seguridad empresarial*
Idioma del trabajo:	*Castellano*
Palabras clave	*DevSecOps Automatización Seguridad*

Resumen:

Este trabajo nace del creciente aumento en popularidad de DevSecOps. DevSecOps integra la seguridad en el proceso DevOps.

El objetivo principal de este trabajo es la automatización de los controles de seguridad en ciertas fases del desarrollo software. En concreto, en este trabajo se automatizarán:

Las pruebas estáticas de seguridad (SAST) mediante controles de calidad del software y la comprobación de vulnerabilidades en las dependencias.

Las pruebas dinámicas de seguridad (DAST) mediante el análisis dinámico de una aplicación web.

La seguridad en la infraestructura mediante el análisis de las vulnerabilidades CVE de las imágenes Dockers.

Para ello en este trabajo se han desarrollado tres proyectos independientes en Jenkins que automatizan cada una de los controles anteriores y permiten conocer tanto la calidad del software como las vulnerabilidades en las dependencias, aplicaciones y en la infraestructura.

En este proyecto se realiza un uso extenso de Dockers, desde la utilización del Docker de Jenkins hasta el uso de las imágenes Dockers de las diferentes herramientas integradas.

Abstract:

This thesis emerge from the popularity increase of DevSecOps. DevSecOps integrates security in the DevOps process.

The main objective of this thesis is the automation of security controls in certain phases of software development. To sum up, in this work will be automate:

The static security tests (SAST) through software quality controls and dependency vulnerabilities checking.

Dynamic security tests (DAST) through the dynamic analysis of a web application.

Infrastructure security by analysing the CVE vulnerabilities of the Dockers images.

To this aim, in this thesis have been developed three independent projects in Jenkins which automate each of the previous controls named and allow knowing both the quality of the software and the vulnerabilities in the dependencies, applications and in the infrastructure.

In this thesis an extensive use of Dockers is made, from the use of Jenkins Docker to the use of the Dockers images of the different tools integrated.

Índice

1. Introducción 1
   1.1 Contexto y justificación del Trabajo 1
   1.2 Objetivos del Trabajo 4
   1.3 Enfoque y método seguido 4
   1.4 Planificación del Trabajo 5
   1.5 Breve sumario de productos obtenidos 7
   1.6 Breve descripción de los otros capítulos de la memoria 7
   1.7 Herramientas 7
   1.8 Arquitectura 8
2. Control de calidad y dependencias del software 10
   2.1 Análisis estático seguro de código (SAST) 10
   2.2 Automatizando SAST con Jenkins 11
3. Automatización de las pruebas dinámicas de seguridad 15
   3.1 Pruebas dinámicas de seguridad (DAST) 15
   3.2 Automatizando DAST con Jenkins 17
4. Automatización de la seguridad en la infraestructura 19
   4.1 Características y securización en Docker 19
   4.2 Control de vulnerabilidades en contenedores 21
5. Conclusiones 25
   5.1 Conclusiones 25
   5.2 Objetivos cumplidos 25
   5.3 Líneas de trabajo futuro 25
6. Glosario 26
7. Bibliografía 27
8. Anexos 30
   8.1 Versiones de las herramientas 30
   8.2 Manual instalación Jenkins 31
   8.3 Manual configuración de proyectos tipo “Pipeline” en Jenkins 33
   8.4 Pipeline control de calidad y dependencias del software 35
   8.6 Instalación y ejecución de Anchore Engine 38
   8.7 Manual de configuración de Docker registry 39
   8.8 Manual configuración del proyecto de automatización de la seguridad en la infraestructura en Jenkins 40

Lista de figuras

Ilustración 1: Etapas clásicas de producción software [1] 1
Ilustración 2: Concepto desarrollo DevSecOps [1] 1
Ilustración 3: Pipeline DevSecOps [5] 2
Ilustración 4: Velocidad de parcheo según frecuencia de escaneos [3] 4
Ilustración 5: Diagrama Gantt - Planificación 6
Ilustración 6: Docker in Docker [30] 9
Ilustración 7: Diagrama pipeline SAST 11
Ilustración 8: Etapas pipeline SAST 12
Ilustración 9: Informe PMD Warning – Unused Import 13
Ilustración 10: Informe PMD Warning – If statements 13
Ilustración 11: Resultado análisis SpotBugs 13
Ilustración 12: Dependency-Check vulnerabilidad Bootstrap 14
Ilustración 13: Pirámide de Cohn [39] 16
Ilustración 14: Cono de helado de pruebas automáticas [39] 16
Ilustración 15: Diagrama pipeline DAST 17
Ilustración 16: ZAP Scanning Report 18
Ilustración 17: Ejemplo ZAP Report 18
Ilustración 18: Docker vs Máquina virtual [51] 19
Ilustración 19: Diagrama pipeline Anchore Engine 22
Ilustración 20: Anchore Engine Report CVE 23
Ilustración 21: Anchore Engine Evaluation 24
Ilustración 22: Contraseña inicial Jenkins 32
Ilustración 23: Jenkins - New Item 33
Ilustración 24: Jenkins - Pipeline project 33
Ilustración 25: Jenkins - Configuración proyecto 34
Ilustración 26: Imágenes Anchore Engine 38
Ilustración 27: Jenkins - Freestyle project 40
Ilustración 28: Jenkins - Repositorio Web 41
Ilustración 29: Pipeline freestyle - Execute Shell 41
Ilustración 30: Pipeline freestyle - Anchore Engine 42

1. Introducción

1.1 Contexto y justificación del Trabajo

En las metodologías de desarrollo de software tradicional, la seguridad opera una vez que se ha diseñado e implementado el sistema, donde los defectos de seguridad son determinados por el personal de seguridad y corregidos por el personal de operaciones antes de que el sistema entre en producción (Ilustración 1). Esta forma de trabajo no solo ralentiza el proceso de entrega de software, sino que también supone un incremento en lo referente a costes. [1]

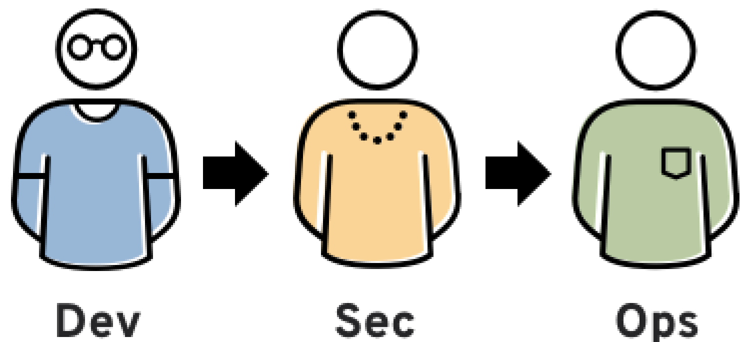


Ilustración 1: Etapas clásicas de producción software [1]

A causa de la creciente demanda empresarial de DevOps, los procesos de securización tradicionales se han convertido en un obstáculo a eliminar, ya que, ahora se busca realizar entregas frecuentes con desarrollos rápidos. Para ello, a la unión de los procesos de desarrollo y producción en DevOps, se le suma el proceso de securización para obtener DevSecOps (Ilustración 2). [1]

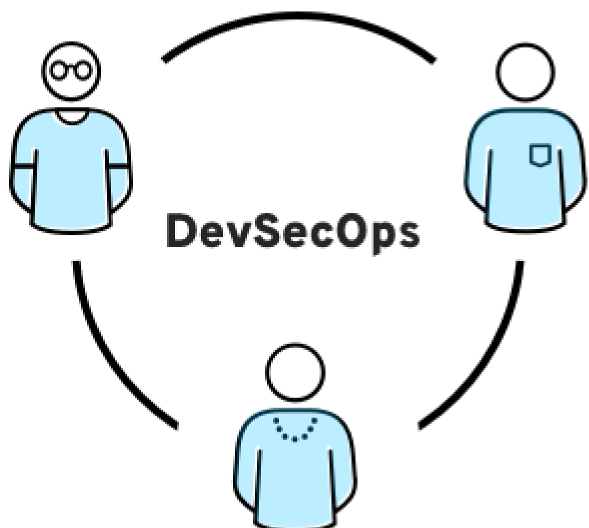


Ilustración 2: Concepto desarrollo DevSecOps [1]

El término DevSecOps describe un ciclo de vida del desarrollo software de entrega continua donde la seguridad se tiene en cuenta en todas sus

fases (Diseño, Desarrollo, Pruebas y Producción) [5]. DevSecOps parte del concepto general y de los valores de DevOps para integrar la seguridad en todos los pasos del proceso de desarrollo (Ilustración 3), de esta manera la seguridad se integra en el desarrollo de manera activa [4].

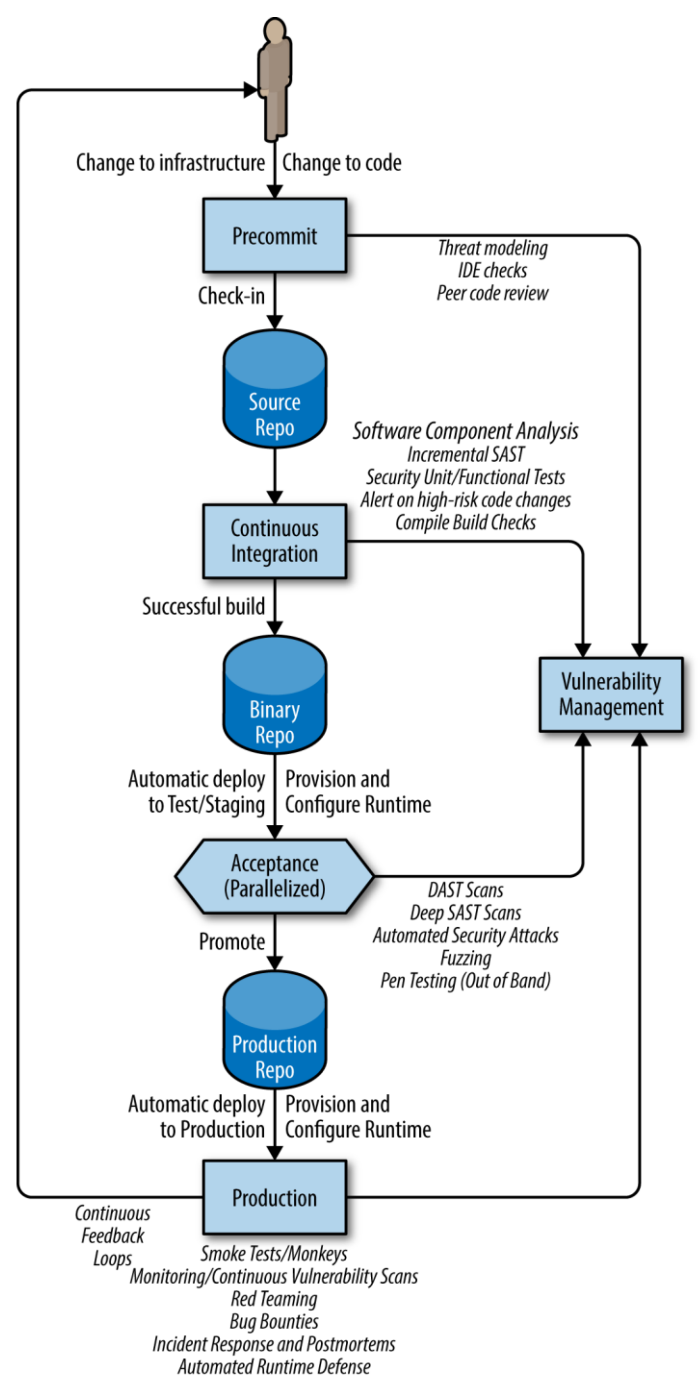


Ilustración 3: Pipeline DevSecOps [5]

Integrar la seguridad en todas las etapas del proceso de desarrollo de manera activa requiere hacer uso de herramientas que permitan la automatización de los controles de seguridad. Estos controles automáticos, deberán comprobar que no existan vulnerabilidades cuando se produzca un cambio en el código o en la infraestructura antes de que entren en producción. Para poder realizar las comprobaciones en la configuración de la infraestructura, esta debe estar bajo herramientas de control de versiones. También se deberán desarrollar test automatizados que tengan en cuenta la detección de fallos de seguridad tanto en la ejecución del software como en la infraestructura. Por otro lado, se deberá llevar un control de las vulnerabilidades de las dependencias externas (frameworks, librerías de terceros…), haciendo uso de herramientas que recojan las vulnerabilidades de estas dependencias. [5]

DevSecOps apuesta la formación del personal en el ámbito de la seguridad, ya que uno de los propósitos de DevSecOps es que la responsabilidad de la seguridad recaiga en todo el grupo, y no solamente en el grupo de seguridad como pasa en los enfoques tradicionales. [2]

DevSecOps no solo ofrece un producto seguro con entregas rápidas. Las organizaciones que aplican correctamente DevSecOps están experimentando un aumento de ingresos y ganancias en torno al 60%, y son hasta 2,4 veces más propensas a hacer crecer su negocio [3]. DevSecOps tiene el potencial para influir positivamente en el estado de la seguridad del software [3]. Otros beneficios que brinda DevSecOps son: velocidad de recuperación, escalabilidad y disminución de amenazas [5].

Según el informe "State of Software Security" de Veracode [3], aún son muchas organizaciones que, aunque implementan DevOps, siguen dejando el proceso de seguridad para el final. Este dato proviene del número de escaneos que han realizado sus clientes a las aplicaciones que tienen en su servicio. Aun así, Veracode ha notado un incremento significativo en el número de escaneos realizados a las aplicaciones almacenadas en su servicio. Según Veracode, la media de escaneos de seguridad de aplicaciones por año es más de 7, aunque existen clientes que han llegado a los 1045 escaneos por año. Estos datos muestran una tendencia positiva al uso de DevSecOps.

Se ha demostrado que existe una gran correlación entre la automatización la seguridad con DevSecOps, y el tiempo que una organización tarda en arreglar las vulnerabilidades. Cuanto más escaneos de seguridad se realicen durante proceso de desarrollo, menos se tardan en reparar las vulnerabilidades. En el siguiente gráfico (Ilustración 4), se observa que las organizaciones que realizan más escaneos de seguridad, arreglan más fallos de seguridad en el mismo

periodo de tiempo que las que realizan menos escaneos de seguridad. [3]

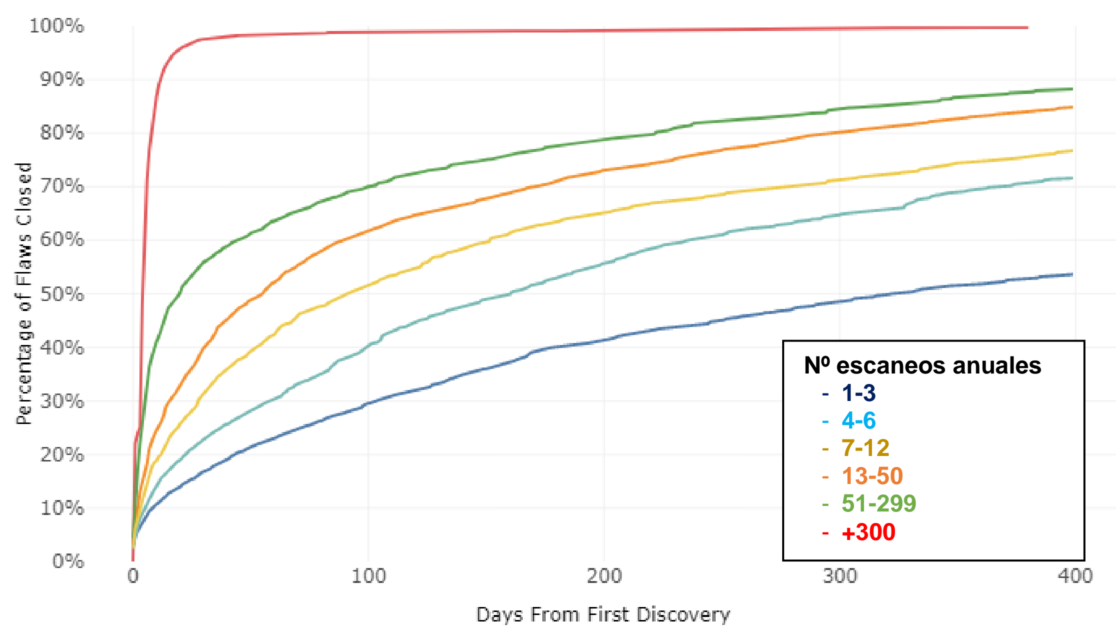


Ilustración 4: Velocidad de parcheo según frecuencia de escaneos [3]

Este trabajo se realiza debido a la creciente demanda de DevSecOps en el mundo empresarial, junto con la gran cohesión que existe entre la seguridad y el desarrollo.

1.2 Objetivos del Trabajo

El objetivo de este trabajo es poner en marcha controles automáticos que permitan la correcta aplicación de DevSecOps resolviendo los problemas que se puedan encontrar con el fin de garantizar la seguridad del software. En concreto, este trabajo se desarrollará:

- La puesta en marcha del control automatizado de calidad del software y de la comprobación de vulnerabilidades en las dependencias de terceros a través de análisis estáticos.
- La puesta en macha de pruebas dinámicas automáticas de seguridad.
- La puesta en marcha de controles de seguridad automáticos de la infraestructura.

1.3 Enfoque y método seguido

En este trabajo se han desarrollado tres pipelines independientes en Jenkins con el fin de cumplir cada uno de los objetivos marcados en el punto anterior.

1.4 Planificación del Trabajo

Nombre de la tarea	Fecha de Inicio	Fecha final
TFM		
Entrega 1 - Plan de trabajo	**20/02/19**	**05/03/19**
Marcado de objetivos	20/02/19	27/02/19
Planificación	27/02/19	27/02/19
Contexto y justificación del trabajo	28/02/19	05/03/19
Entrega 2 - Control de calidad y dependencias del software	**06/03/19**	**02/04/19**
Descripción del problema a resolver	06/03/19	07/03/19
Puesta en marcha y configuración del programa elegido	08/03/19	02/04/19
Entrega 3 - Automatización de las pruebas de seguridad	**03/04/19**	**30/04/19**
Descripción del problema a resolver	03/04/19	04/04/19
Puesta en marcha y configuración del programa elegido	05/04/19	30/04/19
Entrega 4 - Memoria Final	**01/05/19**	**04/06/19**
Automatización de la infraestructura	01/05/19	28/05/19
Descripción del problema a resolver	01/05/19	02/05/19
Puesta en marcha y configuración del programa elegido	03/05/19	28/05/19
Enfoque y método seguido	29/05/19	29/05/19
Conclusión	30/05/19	31/05/19
Abstract	31/05/19	02/06/19
Revisión memoria y corrección de errores	03/06/19	04/06/19
Entrega 5 - Presentación en vídeo	**05/06/19**	**11/06/19**
Realización y entrega del vídeo	05/06/19	11/06/19
Entrega 6 - Defensa del TFM	**17/06/19**	**21/06/19**
Defensa del TFM	17/06/19	21/06/19

Nombre de la tarea	Fecha de Ir	Fecha final
TFM		
Entrega 1 – Plan de trabajo	**20/02/19**	**05/03/19**
Marcado de objetivos	20/02/19	27/02/19
Planificación	27/02/19	27/02/19
Contexto y justificación del trabajo	28/02/19	05/03/19
Entrega 2 – Control de calidad y dependencias del software	**06/03/19**	**02/04/19**
Descripción del problema a resolver	06/03/19	07/03/19
Puesta en marcha y configuración del programa elegido	08/03/19	02/04/19
Entrega 3 – Automatización de las pruebas de seguridad	**03/04/19**	**30/04/19**
Descripción del problema a resolver	03/04/19	04/04/19
Puesta en marcha y configuración del programa elegido	05/04/19	30/04/19
Entrega 4 – Memoria Final	**01/05/19**	**04/06/19**
Automatización de la infraestructura	01/05/19	28/05/19
Descripción del problema a resolver	01/05/19	02/05/19
Puesta en marcha y configuración del programa elegido	03/05/19	28/05/19
Enfoque y método seguido	29/05/19	29/05/19
Conclusión	30/05/19	31/05/19
Abstract	31/05/19	02/06/19
Revisión memoria y corrección de errores	03/06/19	04/06/19
Entrega 5 – Presentación en vídeo	**05/06/19**	**11/06/19**
Realización y entrega del vídeo	05/06/19	11/06/19
Entrega 6 – Defensa del TFM	**17/06/19**	**21/06/19**
Defensa del TFM	17/06/19	21/06/19

Ilustración 5: Diagrama Gantt - Planificación

1.5 Breve sumario de productos obtenidos

En este trabajo se ha obtenido:

- La automatización de pruebas estáticas de seguridad.
- La automatización de pruebas dinámicas de seguridad.
- La automatización del análisis CVE de imágenes Dockers.

1.6 Breve descripción de los otros capítulos de la memoria

En este trabajo se han desarrollado los siguientes capítulos:

- En el capítulo 2 se da solución a la automatización de las pruebas estáticas de seguridad (SAST), automatizando el control de calidad del software y las vulnerabilidades de las dependencias. Para ello se ha utilizado Jenkins con los plugins Warnings Next Generation y OWASP Dependency-check.
- En el capítulo 3 se da solución a la automatización de las pruebas dinámicas de seguridad (DAST), analizando una web en funcionamiento. Para ello se ha utilizado Jenkins con el Docker OWASP ZAP y el plugin HTML Publisher.
- En el capítulo 4 se da solución a la automatización de la seguridad de la infraestructura, analizando las vulnerabilidades CVE de imágenes Dockers. Para ello se ha utilizado Jenkins con el Docker Anchore-engine y el plugin Anchore Container Image Scanner.
- En el capítulo 5 se recogen las conclusiones, los objetivos cumplidos y futuras líneas de trabajo.
- En los anexos se describe cómo configurar el sistema para poder realizar la ejecución de cada pipeline.

1.7 Herramientas

El primer reto planteado durante la realización de este trabajo fue la elección de una herramienta adecuada para poder automatizar los controles de seguridad durante las diferentes etapas de DevSecOps. La herramienta elegida ha fue Jenkins.

Jenkins es un servidor de automatización gratuito y open-source, que puede ser usado para automatizar todas las tareas relacionadas con la compilación, pruebas, y entrega o integración continua [25]. En este trabajo se ha utilizado Jenkins para dar solución a la automatización del control de calidad del software y de las vulnerabilidades de las dependencias (Capítulo 2), de las pruebas dinámicas (Capítulo 3) y de las securización de la infraestructura (Capítulo 4).

Entre las diferentes alternativas de instalación que ofrece Jenkins [26], se escogió el Docker de Jenkins debido a su facilidad a la hora del despliegue (Anexo 8.2).

Para las pruebas dinámicas (Capítulo 3) se ha utilizado el programa ZAP de OWASP [33]. Este programa ofrece un conjunto variado de test que se pueden realizar a una URL [34], entre otras, encontramos:

- Spider: herramienta que se usa para descubrir nuevas URLs desde la URL dada. [42]
- Escáner activo: utiliza ataques conocidos para escanear activamente el objetivo y encontrar vulnerabilidades. [43]
- Escáner pasivo: escanea todos los mensajes HTTP enviados y recibidos de la URL en busca de vulnerabilidades. [44]
- Man in the middel: permite descifrar todos los menajes entre la URL y el cliente. [45]
- Otros: SQL Injection, JavaScript, web sockets. [46]

Para el análisis de las vulnerabilidades CVE de imágenes Dockers (Capítulo 4) se ha utilizado la herramienta Anchore Engine en Jenkins (Anexo 8.8).

Anchore Engine es un proyecto de código abierto que proporciona un servicio centralizado para inspeccionar, analizar y certificar imágenes de contenedores. Anchore Engine permite que los desarrolladores puedan obtener análisis detallados sobre las imágenes Dockers y definir políticas de seguridad para ser aplicadas durante el análisis de imágenes. [48]

1.8 Arquitectura

Este trabajo se ha realizado sobre un sistema host Ubuntu sobre el que corre el contenedor de Jenkins [26] (Anexo 8.1). Este contenedor es el que orquesta la pipeline y, mediante el Docker in Docker, ejecuta los diferentes contenedores necesarios para la ejecución de cada etapa. En el capítulo 4 se han utilizado una máquina extra donde se ejecutaba el registry de Docker (Anexo 8.7).

Docker in Docker permite que el contenedor de Jenkins pueda ejecutar otros contenedores sin necesidad de acceder al host. Esto es posible gracias a la compartición del socket Docker de UNIX entre el host y el contenedor de Jenkins [29] (Anexo 8.2). Realmente, con Docker in Docker, todos los contenedores se ejecutan en el host (Ilustración 6).

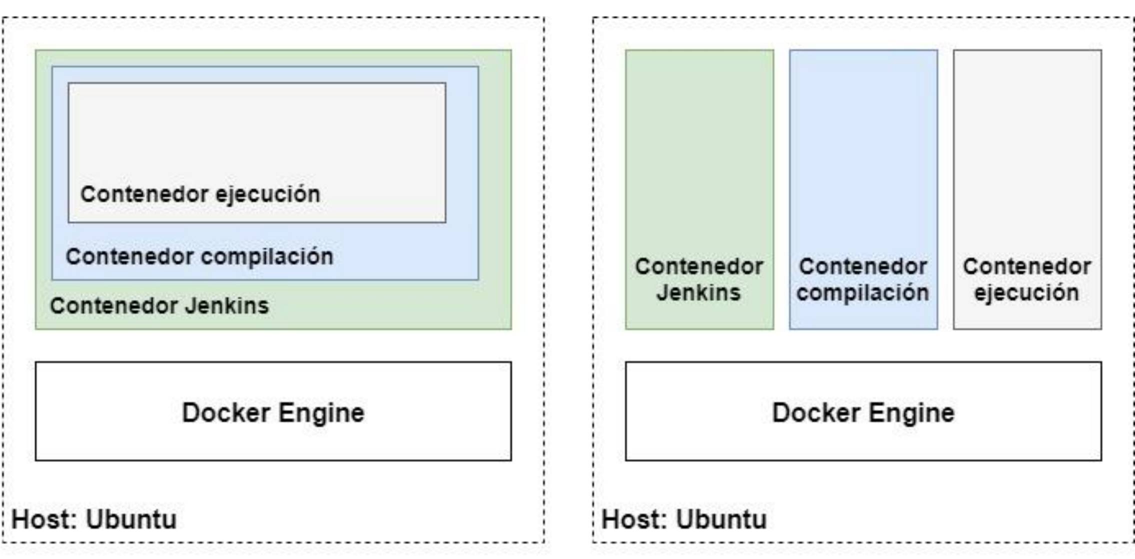


Ilustración 6: Docker in Docker [30]

2. Control de calidad y dependencias del software

En este capítulo se dará solución a la automatización del control de calidad del software y de las vulnerabilidades de las dependencias.

2.1 Análisis estático seguro de código (SAST)

Para dar solución a la automatización del control de calidad del software y de las vulnerabilidades de las dependencias, se ha desarrollado una pipeline en Jenkins (Anexo 8.4).

El control de calidad del software tiene como objetivo identificar los problemas de seguridad en el código fuente. Este control se puede realizar tanto en la fase de precommit, integrándolo en el IDE, como en la fase de integración continua, integrándolo en la pipeline (Ilustración 3). Este control se lleva a cabo mediante las pruebas SAST (Static Analysis Security Testing) cuyo objetivo es examinar el código fuente con el fin de encontrar posibles vulnerabilidades [6].

En este trabajo, se han integrado las pruebas SAST en la pipeline de Jenkins mediante el plugin Warnings Next Generation [8]. Se ha elegido este plugin por ser el más completo que se ha encontrado, esto es debido a que ha integrado otros plugins y herramientas [9], de los cuales se utilizarán en te proyecto para el análisis estático del código:

Plugin/herramienta	Objetivo
Maven Console	Recolectar los errores y avisos durante la compilación del proyecto. [9]
Javadoc	Publicar Javadoc. [9]
Checkstyle	Analizar el estilo del código en referencia al estándar de codificación. [23]
Cpd	Buscar código duplicado. [24]
PMD	Buscar fallos en la programación como variables no utilizadas, bloques catch vacíos o creación innecesaria de objetos, entre otros. [21]
SpotBugs	Buscar bugs en el código. [22]
Task Scanner	Buscar las tareas abiertas marcadas en el código con TODO o TOFIX. [9]

El control de dependencias tiene como objetivo controlar y minimizar las vulnerabilidades provenientes de los frameworks o librerías de terceros. Para ello, se ha integrado en la pipeline el plugin OWASP Dependency-Check [10], cuya 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++ [7].

2.2 Automatizando SAST con Jenkins

El proyecto que se ha analizado ha sido un fork del proyecto Warnings-ng-plugin del repositorio oficial de Jenkins en GitHub [14]. El proyecto versionado [15] ha sido modificado para adaptar la pipeline de Jenkins a las necesidades de este proyecto.

La pipeline se define en el fichero Jenkinsfile del proyecto que, tras configurarlo en Jenkins (Anexo 8.3), se procesa cada vez se ejecuta el proyecto.

En esta pipeline se utiliza el Docker de maven para realizar un análisis estático del código fuente, y publicar el resultado del análisis en Jenkins. Para ello, la pipeline realiza las siguientes acciones (Ilustración 7):

- Actualización del proyecto desde GitHub.
- Ejecución del Jenkinsfile donde:
    - Se compila y se realiza el análisis estático del código.
    - Se analizan las dependencias.
- Finalmente se publica el informe del análisis estático.

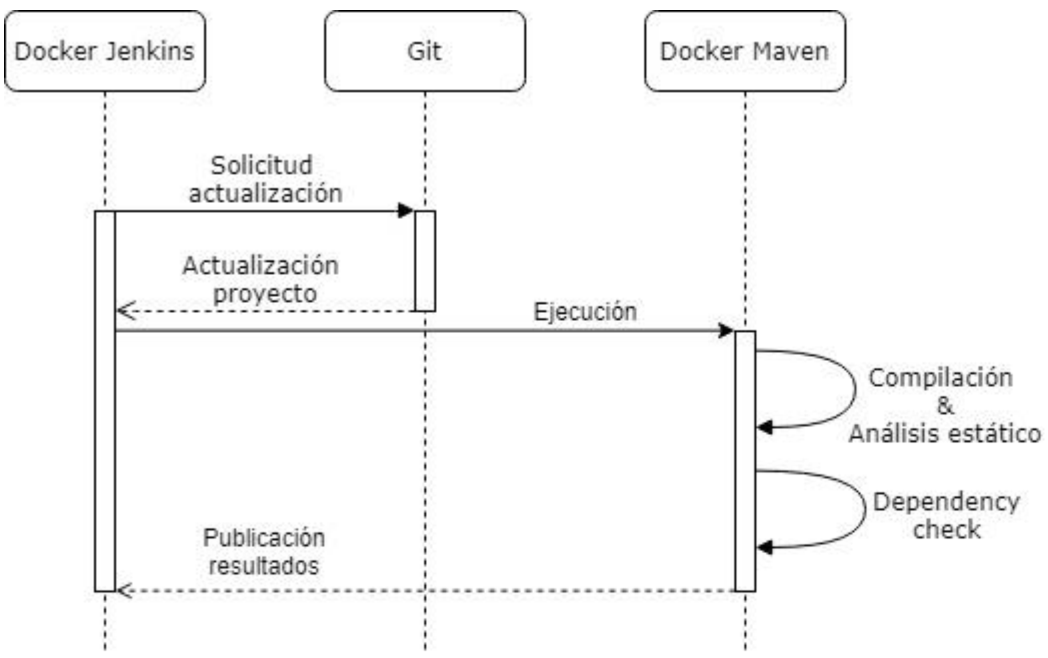


Ilustración 7: Diagrama pipeline SAST

La pipeline definida en este proyecto (Anexo 8.4) se compone de tres partes principales: el agente (agent), las etapas (stages) y las acciones que se realizan al final (post). (Ilustración 8)

Pipeline

Agent: Docker maven:3-alpine

Stage: Build

Stage: Dependency Check

Post-actions

Ilustración 8: Etapas pipeline SAST

En la pipeline, el agente indica donde se ejecutarán las diferentes fases, en este caso es una imagen Docker maven versión 3-alpine.

En la sección de etapas, se definen las diferentes etapas que se ejecutarán durante la pipeline, en este caso son dos etapas, la etapa de compilación y análisis del código, y la etapa de análisis de las dependencias.

Por último, en la acciones post-etapas, se definen qué informes se generarán sobre la calidad del software: bugs, tareas abiertas, código duplicado, vulnerabilidades en las dependencias y warnings generados durante la compilación. Estos informes se verán reflejados en la página del proyecto en Jenkins.

Entre los informes generados destacamos, por ejemplo, el informe del analizador de código PMD [21] donde se refleja un aviso sobre la importación de una librería que no está siendo utilizada (Ilustración 9), o el aviso sobre una mejora en el código como es la disminución de líneas (Ilustración 10).

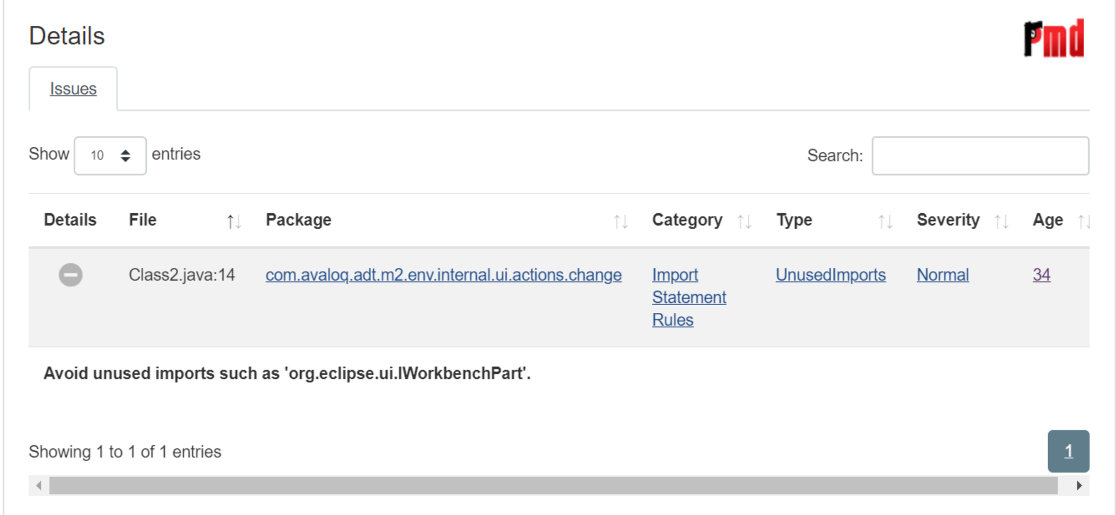


Ilustración 9: Informe PMD Warning – Unused Import

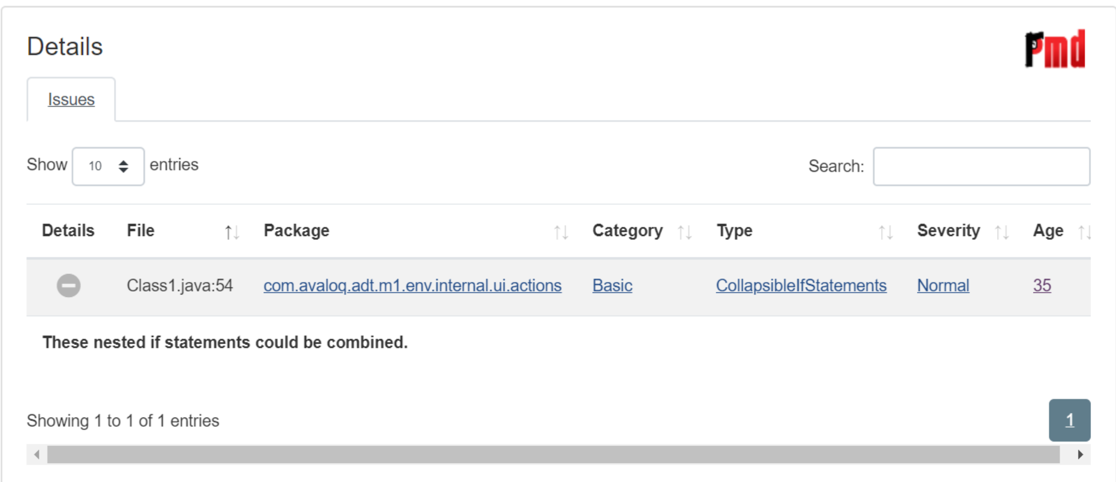


Ilustración 10: Informe PMD Warning – If statements

Otro informe a destacar es el informe del analizador de código SpotBugs [22] que indica que existen dos llamadas a función donde se ignora el valor de retorno. (Ilustración 11)

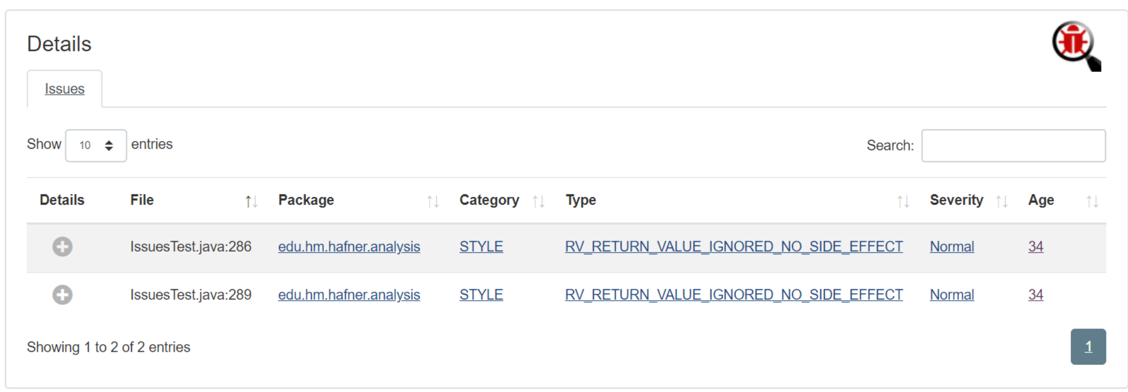


Ilustración 11: Resultado análisis SpotBugs

Durante el análisis de vulnerabilidades realizado por Dependency-Check, se encontró la vulnerabilidad CVE-2019-8331 [12] en la librería Bootstrap (Ilustración 12). Esta vulnerabilidad fue subsanada actualizando los ficheros afectados de la librería Bootstrap a la versión 4.3.1 [13], donde fue corregida esta vulnerabilidad [12]. Al realizar esta corrección, el proyecto pasó a no tener ninguna vulnerabilidad conocida en las dependencias.

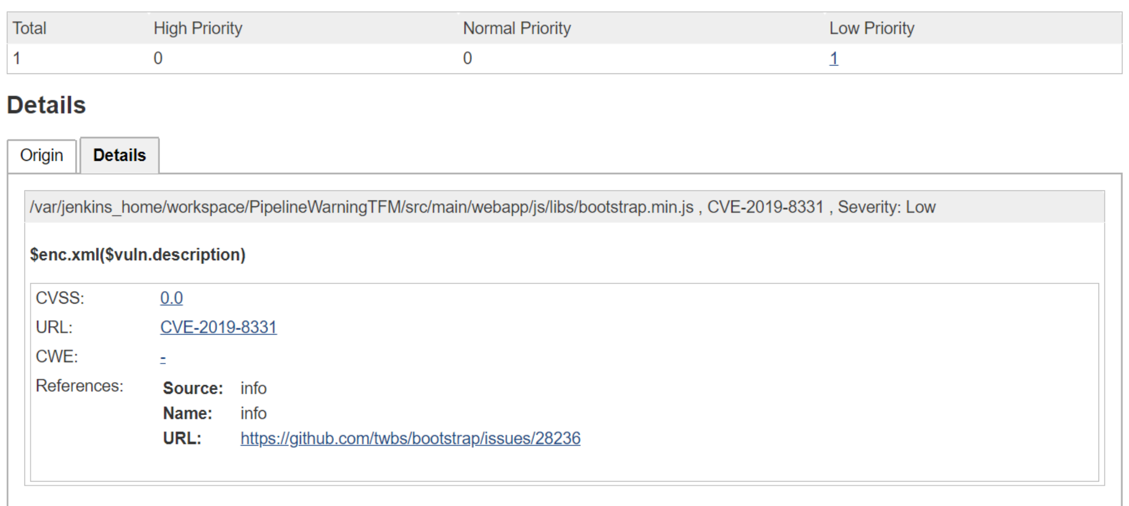


Ilustración 12: Dependency-Check vulnerabilidad Bootstrap

La primera ejecución del chequeo de vulnerabilidades duró unos 13 minutos, si bien, el resto de ejecuciones no superó los 5 segundos, esto es debido a que no se han realizado cambios sustanciales en el proyecto. Si el proyecto tiene previsto un aumento de dependencias que pudiera ralentizar el análisis del código, podría ser viable configurar este control de dependencias como una tarea aparte que se ejecute periódicamente fuera de esta pipeline.

3. Automatización de las pruebas dinámicas de seguridad

En este capítulo se dará solución a la automatización de pruebas dinámicas de seguridad (DAST).

3.1 Pruebas dinámicas de seguridad (DAST)

La automatización de pruebas dinámicas de seguridad (DAST), a diferencia de las pruebas estáticas (SAST), se centran en la búsqueda de vulnerabilidades en tiempo real, es decir, mientras la aplicación se está ejecutando. El fin de las pruebas DAST es de detectar vulnerabilidades no detectadas durante las fases anteriores.

Las pruebas DAST son más difíciles de automatizar que las pruebas SAST [32]. Debido a esto, la automatización de estas pruebas no implica la eliminación por completo de las pruebas manuales, ya que puede haber ocasiones en las que sea más costoso automatizar la prueba que realizarla manualmente [35].

Algunas de las pruebas que se automatizan son, por ejemplo, ciertas pruebas de regresión, o smoke test [35]. Las pruebas de regresión consisten en ejecutar pruebas relevantes, ya ejecutadas con anterioridad, para comprobar que todo lo que funcionaba sigue funcionando correctamente [36]. La finalidad del smoke test es comprobar que las funcionalidades básicas y las críticas siguen funcionando, no se prueba exhaustivamente todo el sistema, solo la parte fundamental [37].

A la hora de automatizar las pruebas, se debe tener en cuenta el grado de automatización de las mismas. La pirámide de pruebas definida por Mike Cohn (Ilustración 13) [38] establece varios niveles de pruebas y señala su grado de automatización [35]:

- Test automáticos unitarios: estos test deben ser los más abundantes ya que es el primer punto para detectar fallos. Un fallo no detectado durante la etapa de desarrollo puede extenderse en las siguientes fases. [35]
- Test automáticos sobre APIs, integraciones y componentes: son los más automatizables y estables. [35]
- Test automáticos sobre la interfaz gráfica: son los test que más cambian debido a que la interfaz gráfica es lo que más suele variar, son lentos de ejecutar y tienen dependencias de otros componentes. [35]
- Test manuales: deben ser los que menos se ejecuten. Los test manuales más repetitivos deben ser automatizados en la medida de lo posible. [35]

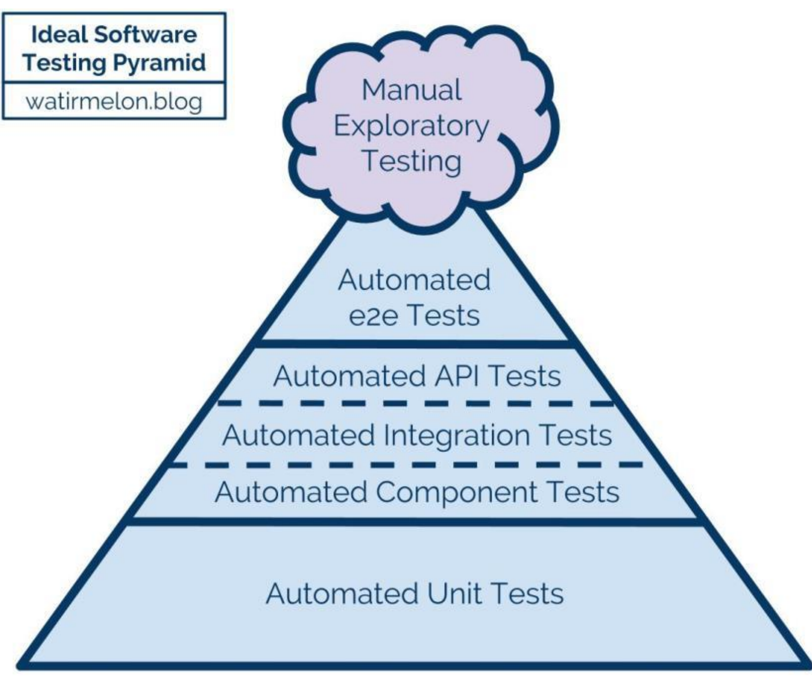


Ilustración 13: Pirámide de Cohn [39]

Durante la automatización de pruebas se debe tener cuidado y tener este concepto siempre en mente (Ilustración 13), ya que una mala estrategia puede desembocar en el denominado "cono de helado" (Ilustración 14). Este "cono de helado" invierte la pirámide de Cohn, realizando una gran cantidad de test manuales y pocos test unitarios y de integración automáticos. [35]

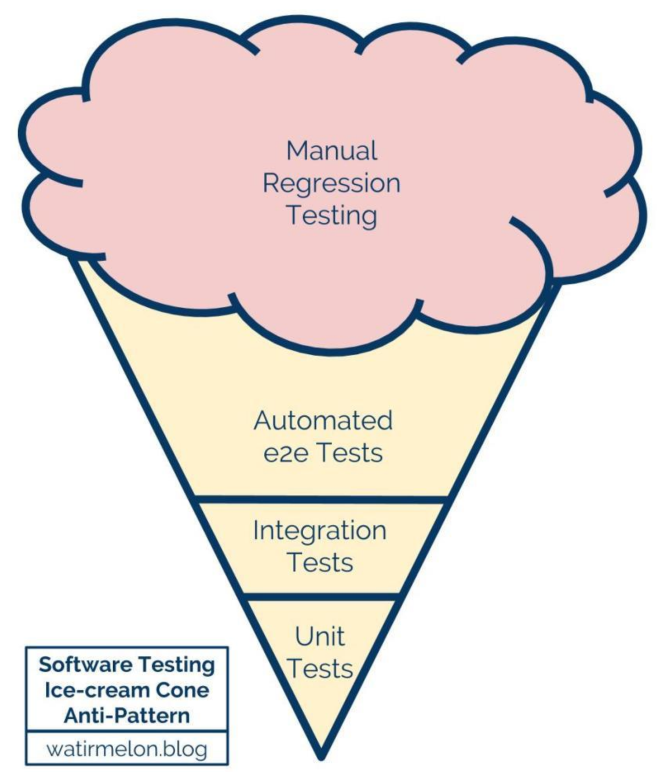


Ilustración 14: Cono de helado de pruebas automáticas [39]

Debido a que la ejecución de las pruebas DAST consumen mucho tiempo, puede ser difícil su integración en las compilaciones diarias. Un enfoque alternativo a ejecutar las pruebas DAST diariamente es ejecutarlas periódicamente, semanalmente o cada quincena. Otro enfoque podría ser escanear áreas críticas diariamente y la aplicación completa periódicamente. [32]

3.2 Automatizando DAST con Jenkins

La automatización de las pruebas dinámicas de seguridad se ha definido como una pipeline diferente a la del análisis estático (Anexo 8.5). En esta pipeline se utiliza el Docker de la herramienta ZAP [40] para realizar un análisis dinámico a una web, y se publica el resultado del análisis en Jenkins. En este caso analizaremos la dirección del propio Docker donde se ejecuta Jenkins, pero en un caso real se podría realizar el análisis sobre una dirección web real. Las acciones que realiza esta pipeline son (Ilustración 15):

- Actualización del proyecto desde GitHub.
- Ejecución del Jenkinsfile donde:
    - Se ejecuta el Docker ZAP, que analiza dinámicamente la web de Jenkins.
- Finalmente se publica el informe del análisis dinámico. Este informe se copia desde el interior del Docker ZAP a la ruta de le ejecución de Jenkins para poder ser publicado.

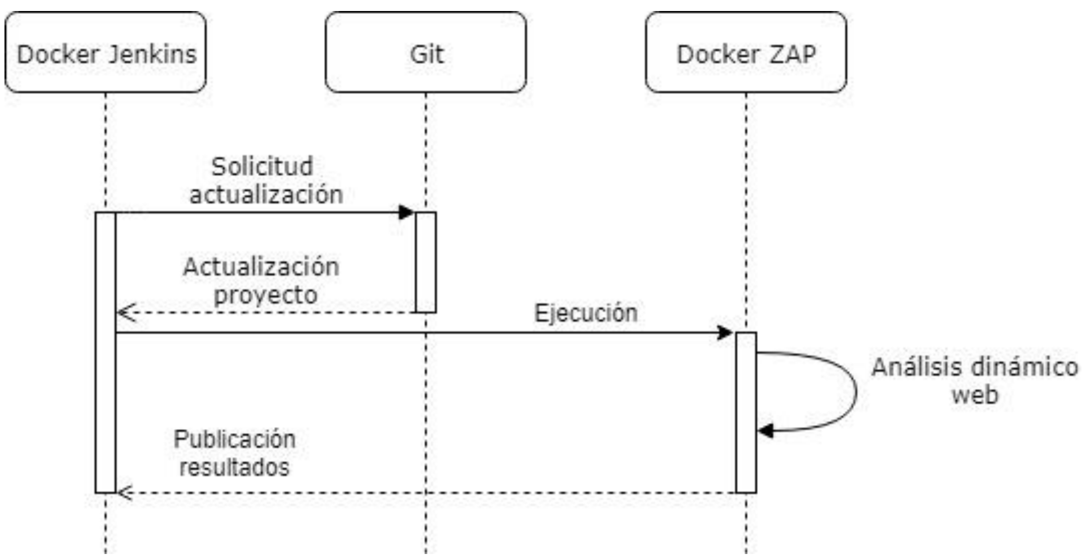


Ilustración 15: Diagrama pipeline DAST

El Docker de ZAP analiza el objetivo con la herramienta Spider [42] cuya finalidad es encontrar nuevas URLs de ese sitio web. La herramienta Spider parte de la URL original y, concatenándole cadenas predefinidas, analiza la respuesta identificando las URLs existentes. Estas URLs son almacenadas en el informe que se genera.

En la pipeline (Anexo 8.5) se distinguen dos etapas, la etapa “Build”, donde se ejecuta el análisis, se genera un informe en formato HTML y se copia a la ruta de la tarea correspondiente dentro del Docker de Jenkins, y la post-etapa, donde se publica este informe en la en la tarea correspondiente. Para la publicación de este informe se ha utilizado el plugin “HTML Publisher” [41], que permite publicar informes HTML relacionándolo con la ejecución en Jenkins.

El informe publicado, agrupa las alertas según su nivel de riesgo (alto, medio, bajo e información). En este caso, en el análisis de la web de Jenkins 5 riesgos bajos y 5 riesgos de información (Ilustración 16).

ZAP Scanning Report

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	0
Low	5
Informational	5

Ilustración 16: ZAP Scanning Report

Este informe también ofrece una información más detallada sobre las vulnerabilidades encontradas, junto a posibles soluciones. Un ejemplo de riesgo bajo es la ausencia de tokens anti-CSRF (Cross Site Request Forgery) en el formulario de envío HTML (Ilustración 17). Lo que puede provocar peticiones ilegítimas con el fin de robar información.

Low (Medium)	Absence of Anti-CSRF Tokens
Description	No Anti-CSRF tokens were found in a HTML submission form. A cross-site request forgery is an attack that involves forcing a victim to send an HTTP request to a target destination without their knowledge or intent in order to perform an action as the victim. The underlying cause is application functionality using predictable URL/form actions in a repeatable way. The nature of the attack is that CSRF exploits the trust that a web site has for a user. By contrast, cross-site scripting (XSS) exploits the trust that a user has for a web site. Like XSS, CSRF attacks are not necessarily cross-site, but they can be. Cross-site request forgery is also known as CSRF, XSRF, one-click attack, session riding, confused deputy, and sea surf. CSRF attacks are effective in a number of situations, including: * The victim has an active session on the target site. * The victim is authenticated via HTTP auth on the target site. * The victim is on the same local network as the target site. CSRF has primarily been used to perform an action against a target site using the victim's privileges, but recent techniques have been discovered to disclose information by gaining access to the response. The risk of information disclosure is dramatically increased when the target site is vulnerable to XSS, because XSS can be used as a platform for CSRF, allowing the attack to operate within the bounds of the same-origin policy.
URL	http://192.168.1.170:8080/loginError
Method	GET
Evidence	<form method="post" name="login" action="j_acegi_security_check">
URL	http://192.168.1.170:8080/login?from=%2F
Method	GET
Evidence	<form method="post" name="login" action="j_acegi_security_check">
URL	http://192.168.1.170:8080/login?from=%2Fsitemap.xml
Method	GET
Evidence	<form method="post" name="login" action="j_acegi_security_check">
Instances	3

Ilustración 17: Ejemplo ZAP Report

4. Automatización de la seguridad en la infraestructura

En este capítulo se dará solución a la automatización de la securización de la infraestructura.

4.1 Características y securización en Docker

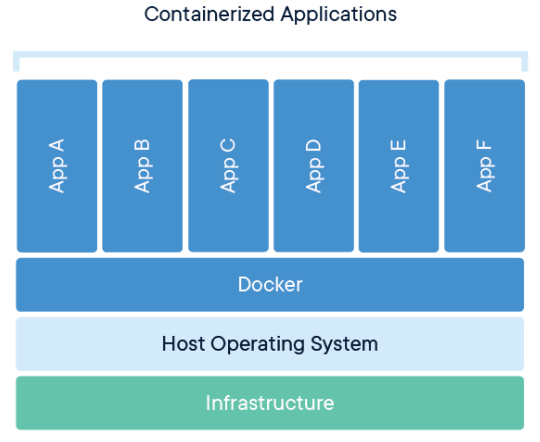
Contenedores como LXC, rtk y Docker han sido introducidos en DevSecOps [6].

La tecnología de contenedores, como es el caso de Docker, hacen que sea mucho más fácil para los desarrolladores implementar una aplicación y empaquetarla junto con las dependencias requeridas [6].

Docker es una herramienta de código abierto que facilita la creación, el despliegue y la ejecución de aplicaciones a través de contenedores [51]. Un contenedor es un sistema en el que se ejecutan de forma aislada un conjunto de procesos [50].

Los contenedores permiten a los desarrolladores empaquetar una aplicación junto con sus dependencias y librerías, y asegurar que se podrá ejecutar en otro entorno con una configuración diferente. [51]

Docker, a diferencia de las máquinas virtuales, comparte el kernel con el host cargando en memoria solo las librerías necesarias para su ejecución. (Ilustración 18)



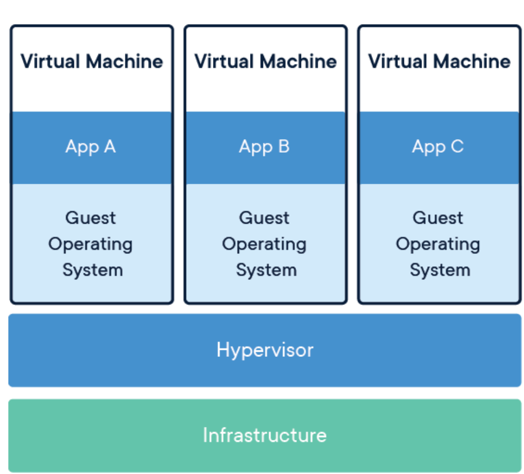


Ilustración 18: Docker vs Máquina virtual [51]

Los principales componentes de un sistema de contenedores Docker son [52]:

- Repositorio: Registro donde se almacenan las imágenes creadas.
- Imagen: Sistema operativo base que puede contener una aplicación lista para ser ejecutada.
- Contenedor: Instancia de una imagen en ejecución.

Las principales características y funcionalidades de los Docker son [52]:

- Autogestión de los contenedores.
- Fiabilidad.
- Compatibilidad con múltiples sistemas, permitiendo el despliegue de los contenedores en múltiples plataformas. Esto se debe a la independencia que existe en el entre el host y los contenedores.
- Compartición de recursos, lo que permite desplegar multitud de contenedores en un mismo equipo físico.
- Despliegue rápido.
- Contenedores ligeros: facilitan su almacenaje, transporte y despliegue.
- Capacidad de ejecutar casi cualquier aplicación.
- La aplicación base de Docker es quien gestiona los recursos existentes y los asigna responsablemente entre los contenedores en ejecución.

Entre las principales ventajas que aporta el uso de contenedores a la CI/CD, destaca la portabilidad de los mismos durante las diferentes etapas (desarrollo, prueba y producción), el control de versiones y la fiabilidad. [50]

José Manuel Ortega, define en su libro "DOCKER. Seguridad y monitorización en contenedores e imágenes" [57] una checklist para para comprobar la seguridad en los contenedores Docker [58]:

- No escribir secretos en los Dockerfile.
- Crear un usuario.
- Remover permisos innecesarios.
- Descargar segura de paquetes.
- No descargar paquetes innecesarios.
- Usar el comando COPY en vez de ADD.
- Utilizar el comando HEALTHCHECK. Este comando sirve para saber que realmente el contenedor se está ejecutando correctamente [59].
- Utilizar el comando "gosu" en lugar de "sudo" cuando sea posible.
- Habilitar Docker Content Trust para protegerse frente a imágenes no oficiales.
- Asegurarse que las imágenes están libres de vulnerabilidades conocidas.
- Escanear frecuentemente las imágenes.
- Actualizar tanto la imagen como los paquetes.

4.2 Control de vulnerabilidades en contenedores

Para automatizar el control de vulnerabilidades en los contenedores, se ha integrado Anchore Engine en Jenkins (Anexo 8.8).

El proyecto configurado en Jenkins (Anexo 8.8) utiliza el plugin “Anchore Container Image Scanner” [49] para realizar el análisis de vulnerabilidades de una imagen Docker. Esta imagen se basa en una imagen Apache HTTP Server [53] donde se copia un proyecto web de Cloud Academy [47].

La pipeline realiza las siguientes acciones (Ilustración 19):

- Actualización del proyecto web [47] desde GitHub.
- Creación del Dockerfile. En este Dockerfile se establece que la imagen base es “httpd:latest” y se realiza una copia del proyecto web actualizado.
- Creación de la imagen donde el nombre tiene el formato {IP_REGISTRY}:{PUERTO_REGISTRY}/web:{FECHA_ACTUAL}.
- Subida de la imagen creada al Registry.
- Ejecución de Anchore Engine:
    - Descarga de la imagen desde el Registry.
    - Análisis de la imagen Docker.
    - Publicación de resultados.
- El proyecto marcará la ejecución de Jenkins fallida si la evaluación falla o existe algún error grave en las imágenes analizadas. Tal y como viene especificado en su configuración (Ilustración 30).

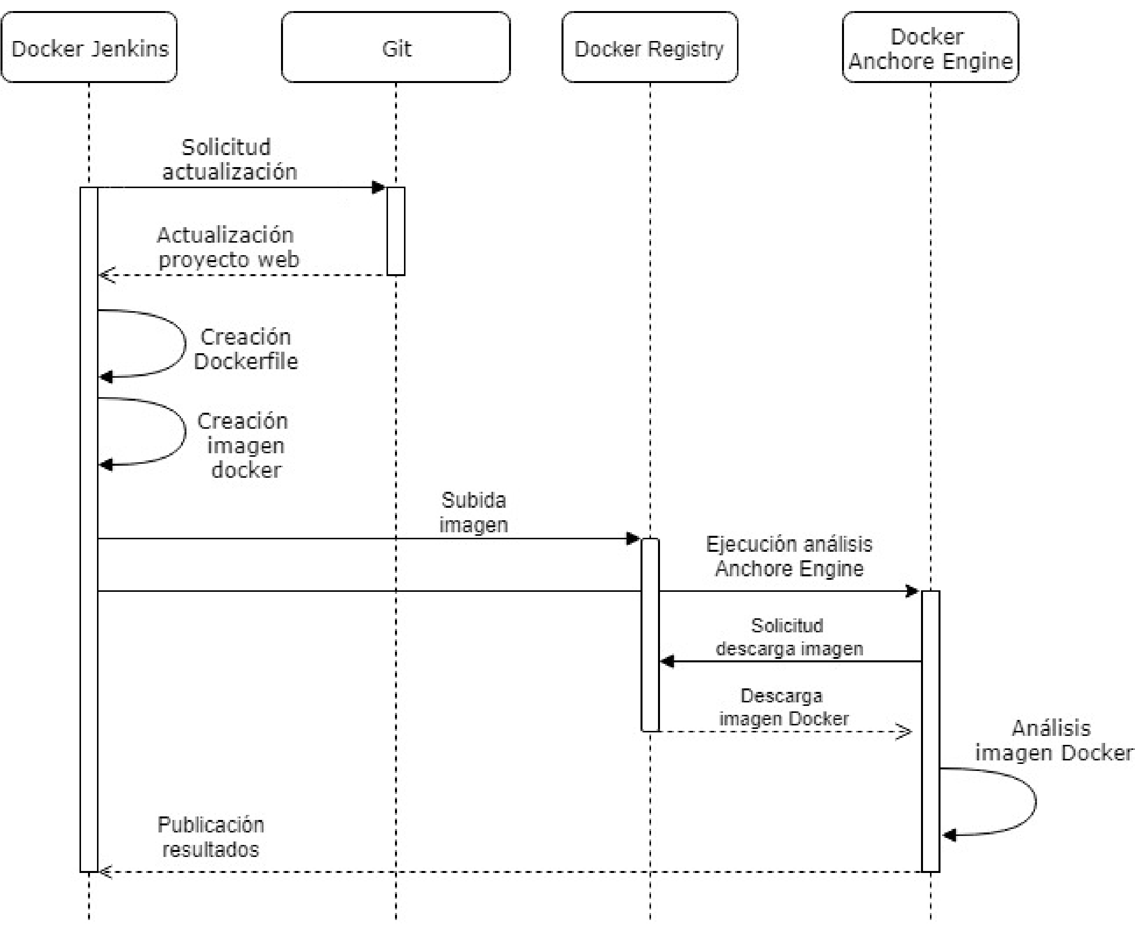
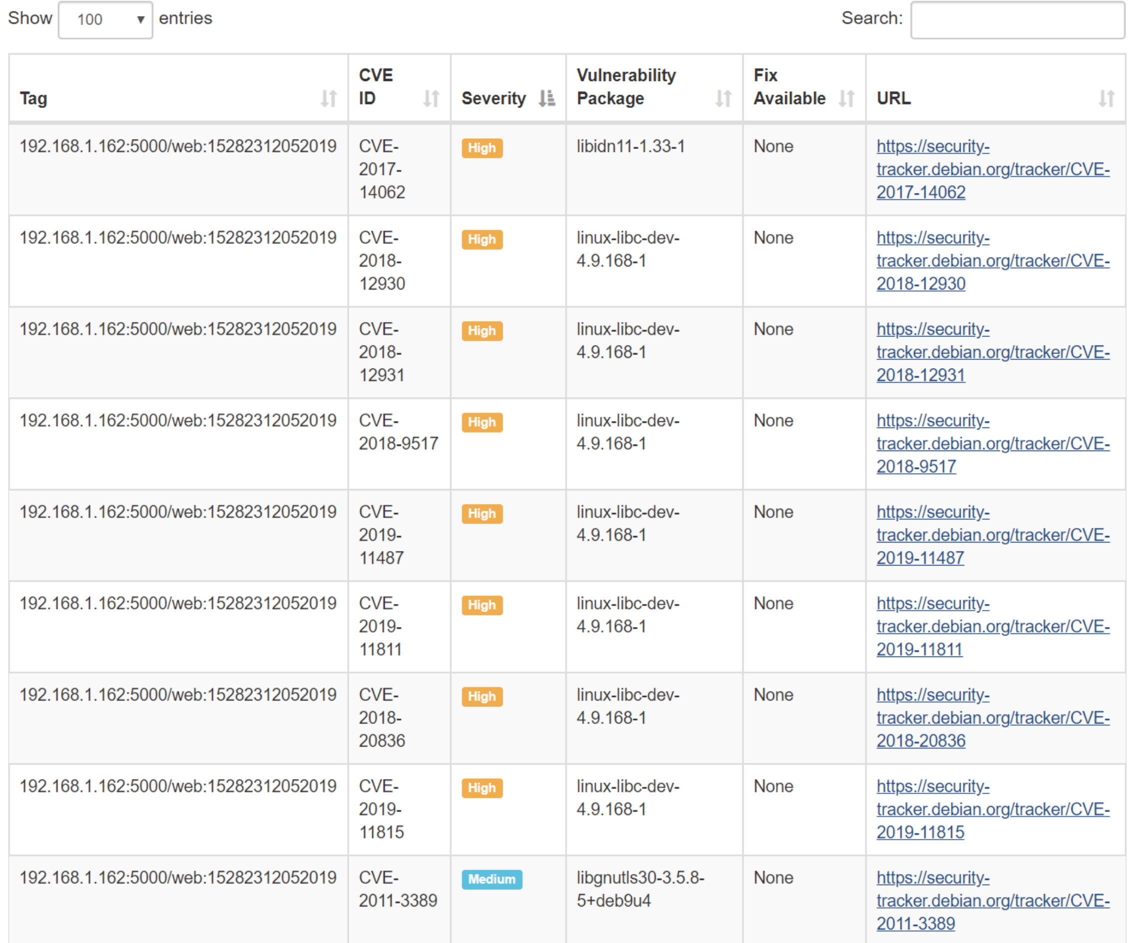


Ilustración 19: Diagrama pipeline Anchore Engine

Tras el análisis de la imagen Docker, el plugin de Anchore Engine publica una lista con las vulnerabilidades CVE encontradas (Ilustración 20). Estas vulnerabilidades están categorizadas por gravedad (alta, media, baja, despreciable y desconocida) y ofrecen información sobre si existe una solución disponible.



Common Vulnerabilities and Exposures (CVE) List

Show 100 entries Search:

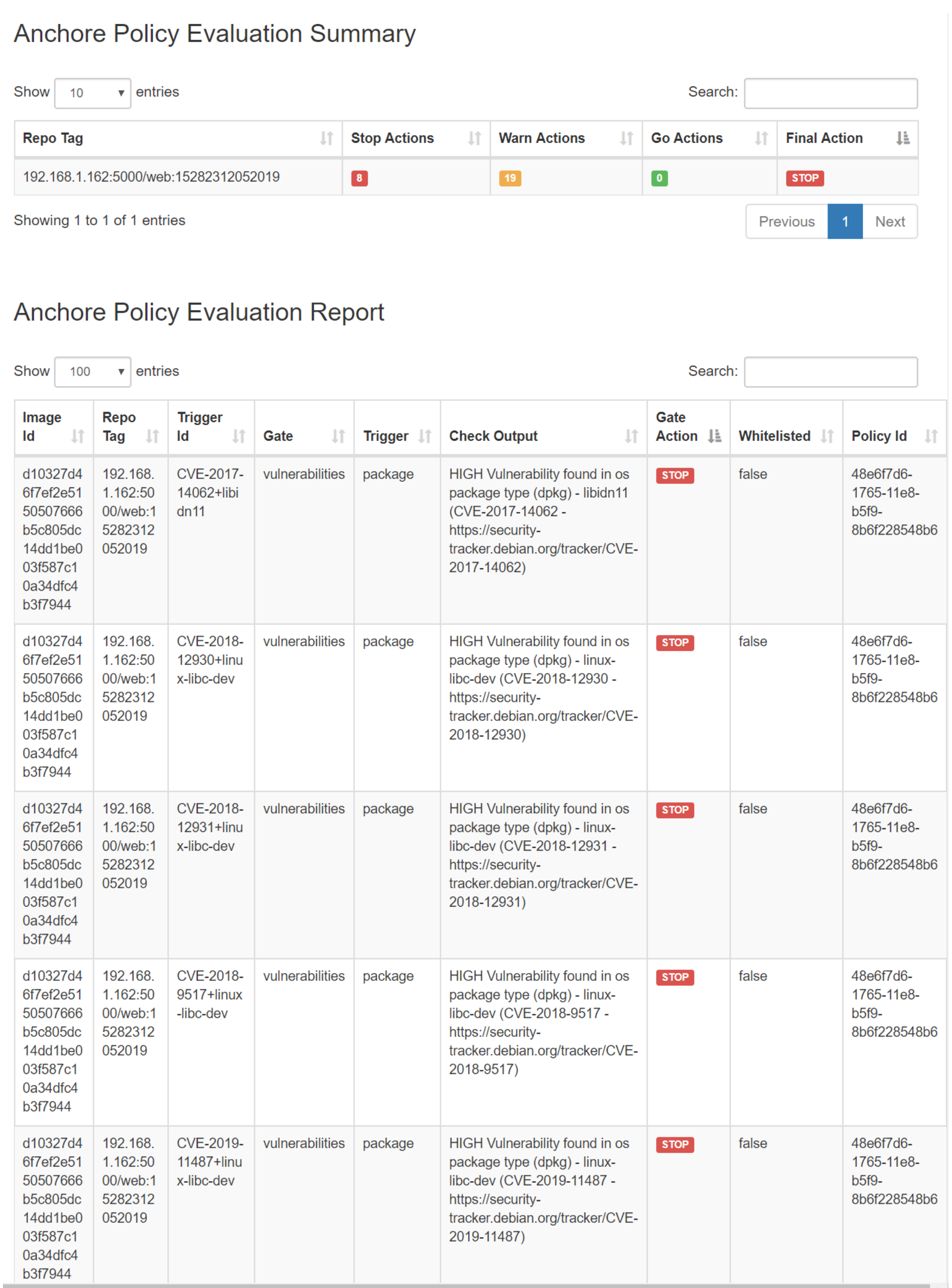
Tag	CVE ID	Severity	Vulnerability Package	Fix Available	URL
192.168.1.162:5000/web:15282312052019	CVE-2017-14062	High	libidn11-1.33-1	None	https://security-tracker.debian.org/tracker/CVE-2017-14062
192.168.1.162:5000/web:15282312052019	CVE-2018-12930	High	linux-libc-dev-4.9.168-1	None	https://security-tracker.debian.org/tracker/CVE-2018-12930
192.168.1.162:5000/web:15282312052019	CVE-2018-12931	High	linux-libc-dev-4.9.168-1	None	https://security-tracker.debian.org/tracker/CVE-2018-12931
192.168.1.162:5000/web:15282312052019	CVE-2018-9517	High	linux-libc-dev-4.9.168-1	None	https://security-tracker.debian.org/tracker/CVE-2018-9517
192.168.1.162:5000/web:15282312052019	CVE-2019-11487	High	linux-libc-dev-4.9.168-1	None	https://security-tracker.debian.org/tracker/CVE-2019-11487
192.168.1.162:5000/web:15282312052019	CVE-2019-11811	High	linux-libc-dev-4.9.168-1	None	https://security-tracker.debian.org/tracker/CVE-2019-11811
192.168.1.162:5000/web:15282312052019	CVE-2018-20836	High	linux-libc-dev-4.9.168-1	None	https://security-tracker.debian.org/tracker/CVE-2018-20836
192.168.1.162:5000/web:15282312052019	CVE-2019-11815	High	linux-libc-dev-4.9.168-1	None	https://security-tracker.debian.org/tracker/CVE-2019-11815
192.168.1.162:5000/web:15282312052019	CVE-2011-3389	Medium	libgnutls30-3.5.8-5+deb9u4	None	https://security-tracker.debian.org/tracker/CVE-2011-3389

Ilustración 20: Anchore Engine Report CVE

Por otro lado, el plugin de Anchore Engine también publica un informe donde aplica las políticas de evaluación a las vulnerabilidades encontradas (Ilustración 21).

Las políticas en Anchore Engine describen qué comprobaciones deben realizarse y cómo deben ser interpretados los resultados. Una política se expresa como un conjunto de reglas que se utilizan para realizar una evaluación de la imagen de un contenedor. Las reglas pueden definir controles como [60]:

- Vulnerabilidades de seguridad.
- Listas blancas y listas negras de paquetes.
- Contenido de un archivo de configuración.
- Presencia de credenciales en una imagen.
- Cambios en el manifiesto de la imagen.
- Puertos expuestos.



Anchore Policy Evaluation Summary

Show 10 entries Search:

Repo Tag	Stop Actions	Warn Actions	Go Actions	Final Action
192.168.1.162:5000/web:15282312052019	8	19	0	STOP

Showing 1 to 1 of 1 entries

Previous 1 Next

Anchore Policy Evaluation Report

Show 100 entries Search:

Image Id	Repo Tag	Trigger Id	Gate	Trigger	Check Output	Gate Action	Whitelisted	Policy Id
d10327d46f7ef2e5150507666b5c805dc14dd1be003f587c10a34dfc4b3f7944	192.168.1.162:5000/web:15282312052019	CVE-2017-14062+libidn11	vulnerabilities	package	HIGH Vulnerability found in os package type (dpkg) - libidn11 (CVE-2017-14062 - https://security-tracker.debian.org/tracker/CVE-2017-14062)	STOP	false	48e6f7d6-1765-11e8-b5f9-8b6f228548b6
d10327d46f7ef2e5150507666b5c805dc14dd1be003f587c10a34dfc4b3f7944	192.168.1.162:5000/web:15282312052019	CVE-2018-12930+linux-libc-dev	vulnerabilities	package	HIGH Vulnerability found in os package type (dpkg) - linux-libc-dev (CVE-2018-12930 - https://security-tracker.debian.org/tracker/CVE-2018-12930)	STOP	false	48e6f7d6-1765-11e8-b5f9-8b6f228548b6
d10327d46f7ef2e5150507666b5c805dc14dd1be003f587c10a34dfc4b3f7944	192.168.1.162:5000/web:15282312052019	CVE-2018-12931+linux-libc-dev	vulnerabilities	package	HIGH Vulnerability found in os package type (dpkg) - linux-libc-dev (CVE-2018-12931 - https://security-tracker.debian.org/tracker/CVE-2018-12931)	STOP	false	48e6f7d6-1765-11e8-b5f9-8b6f228548b6
d10327d46f7ef2e5150507666b5c805dc14dd1be003f587c10a34dfc4b3f7944	192.168.1.162:5000/web:15282312052019	CVE-2018-9517+linux-libc-dev	vulnerabilities	package	HIGH Vulnerability found in os package type (dpkg) - linux-libc-dev (CVE-2018-9517 - https://security-tracker.debian.org/tracker/CVE-2018-9517)	STOP	false	48e6f7d6-1765-11e8-b5f9-8b6f228548b6
d10327d46f7ef2e5150507666b5c805dc14dd1be003f587c10a34dfc4b3f7944	192.168.1.162:5000/web:15282312052019	CVE-2019-11487+linux-libc-dev	vulnerabilities	package	HIGH Vulnerability found in os package type (dpkg) - linux-libc-dev (CVE-2019-11487 - https://security-tracker.debian.org/tracker/CVE-2019-11487)	STOP	false	48e6f7d6-1765-11e8-b5f9-8b6f228548b6

Ilustración 21: Anchore Engine Evaluation

5. Conclusiones

En este capítulo se recogerán las conclusiones obtenidas tras el desarrollo de este trabajo, los objetivos cumplidos y posibles líneas de trabajo futuras.

5.1 Conclusiones

La realización del trabajo demuestra que es posible automatizar los controles de seguridad durante diferentes etapas del desarrollo software.

El desarrollo de este trabajo me ha aportado un nuevo punto de vista sobre la automatización de tareas recurrentes disminuyendo los errores cometidos por los humanos.

5.2 Objetivos cumplidos

Tras el desarrollo de este trabajo se han logrado cumplir los tres objetivos expuestos al inicio del mismo:

- Automatización de las pruebas estáticas de seguridad mediante controles de calidad del software y la comprobación de vulnerabilidades en las dependencias.
- Automatización de las pruebas dinámicas de seguridad mediante el análisis dinámico de una aplicación web. En este caso, aunque se ha cumplido el objetivo, es cierto que no se ha podido realizar una mayor investigación debido a las dificultades encontradas a la hora de poner en marcha las pruebas dinámicas en Jenkins.
- Automatización de la seguridad de la infraestructura mediante el análisis de las vulnerabilidades CVE de las imágenes Dockers.

5.3 Líneas de trabajo futuro

Se proponen las siguientes líneas de trabajo a desarrollar en el futuro:

- Ampliar las pruebas dinámicas.
- Introducir el análisis estático del código durante la etapa de desarrollo.
- Automatizar la securización del resto de fases del desarrollo software.
- Establecer criterios de calidad del software.
- Realizar el desarrollo de una pipeline completa.
- Añadir políticas de seguridad en Anchore Engine.

6. Glosario

CI/CD: Continuous integration/continuous delivery.

CVE: Common Vulnerabilities and Exposures [61].

Cloud Academy: empresa dedicada a la formación en multi-cloud [62].

DAST: Dynamic Application Security Testing.

Dockerfile: fichero de definición de una imagen Docker [59].

GitHub: plataforma de desarrollo colaborativo [63].

IDE: Integrated Development Environment [64].

OWASP: Open Web Application Security Project [65].

SAST: Static Application Security Testing.

CSRF: Cross Site Request Forgery [66].

Checklist: lista de control.

Fork: Bifurcación o copia de un proyecto [67].

Frameworks: conjunto de conceptos, prácticas y criterios utilizados como referencia para resolver un problema [70].

Pipeline: es una nueva forma de trabajar en el mundo DevSecOps con la que se puede definir el ciclo de vida completo de una aplicación. [11]

Registry de Docker: lugar de almacenamiento de imágenes Dockers [68].

Socket UNIX: permite la comunicación entre procesos [69].

7. Bibliografía

[1] https://www.redhat.com/es/topics/devops/what-is-devsecops [26 de febrero de 2019].
[2] https://www.devsecops.org/blog/2015/2/15/what-is-devsecops [26 de febrero de 2019].
[3] Chris Eng, State of software security, Veracode, 35 – 39, Volúmen 9, 2018
[4] https://es.atlassian.com/continuous-delivery/principles/devsecops [27 de febrero de 2019].
[5] Francois Raynaud, DevSecOps Whitepaper – The business benefits and best practices of DevSecOps implementation, DevSecCon, 20, 2017.
[6] Jim Bird, DevOpsSec - Delivering Secure Software through Continuous Delivery, O'REILLY, California, 2016.
[7] https://www.owasp.org/index.php/OWASP_Dependency_Check [17 de marzo de 2019]
[8] https://plugins.jenkins.io/warnings-ng [26 de marzo de 2019]
[9] https://github.com/jenkinsci/warnings-ng-plugin/blob/master/doc/Documentation.md [26 de marzo de 2019]
[10] https://plugins.jenkins.io/dependency-check-jenkins-plugin [26 de marzo de 2019]
[11] https://sdos.es/blog/la-integracion-continua-actual-pasa-por-pipelines [27 de marzo de 2019]
[12] https://nvd.nist.gov/vuln/detail/CVE-2019-8331 [27 de marzo de 2019]
[13] https://github.com/twbs/bootstrap/releases/tag/v4.3.1 [27 de marzo de 2019]
[14] https://github.com/jenkinsci/warnings-ng-plugin [20 de marzo de 2019]
[15] https://github.com/jjcquintero/warnings-ng-plugin [27 de marzo de 2019]
[16] https://docs.docker.com/install/linux/docker-ce/ubuntu/ [25 de marzo de 2019]
[17] https://jenkins.io/doc/book/pipeline/syntax/ [20 de marzo de 2019]
[18] https://jenkins.io/doc/book/pipeline/ [20 de marzo de 2019]
[19] https://issues.jenkins-ci.org/browse/JENKINS-37437 [20 de marzo de 2019]
[20] https://jenkins.io/doc/book/pipeline/docker/ [20 de marzo de 2019]
[21] https://pmd.github.io/ [30 de marzo de 2019]
[22] https://spotbugs.github.io/ [30 de marzo de 2019]
[23] http://checkstyle.sourceforge.net/ [01 de abril de 2019]
[24] https://pmd.github.io/latest/pmd_userdocs_cpd.html [01 de abril de 2019]
[25] https://jenkins.io/doc/ [24 de abril de 2019]
[26] https://jenkins.io/download/ [24 de abril de 2019]
[27] https://hub.docker.com/_/maven/ [24 de abril de 2019]
[28] https://github.com/zaproxy/zaproxy/wiki/Docker [24 de abril de 2019]
[29] https://stackoverflow.com/questions/35110146/can-anyone-explain-docker-sock [24 de abril de 2019]
[30] https://stackoverflow.com/questions/31381322/docker-in-docker-cannot-mount-volume [24 de abril de 2019]

[31] https://techbeacon.com/security/6-devsecops-best-practices-automate-early-often [24 de abril de 2019]
[32] https://dzone.com/articles/the-three-pillars-of-devsecops [24 de abril de 2019]
[33] https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project [24 de abril de 2019]
[34] https://www.owasp.org/index.php/Appendix_A:_Testing_Tools [24 de abril de 2019]
[35] https://www.javiergarzas.com/2015/01/automatizacion-pruebas.html [24 de abril de 2019]
[36] http://www.javiergarzas.com/2014/06/pruebas-de-regresion.html [24 de abril de 2019]
[37] http://www.javiergarzas.com/2014/06/smoke-test-en-menos-de-10-min.html [24 de abril de 2019]
[38] Mike Cohn, Succeeding with Agile, Addison-Wesley Professional, Boston, 2019.
[39] https://watirmelon.blog/testing-pyramids/ [24 de abril de 2019]
[40] https://github.com/zaproxy/zaproxy/wiki/ZAP-Baseline-Scan [24 de abril de 2019]
[41] https://wiki.jenkins.io/display/JENKINS/HTML+Publisher+Plugin [24 de abril de 2019]
[42] https://github.com/zaproxy/zap-core-help/wiki/HelpStartConceptsSpider [24 de abril de 2019]
[43] https://github.com/zaproxy/zap-core-help/wiki/HelpStartConceptsAscan [24 de abril de 2019]
[44] https://github.com/zaproxy/zap-core-help/wiki/HelpStartConceptsPscan [24 de abril de 2019]
[45] https://github.com/zaproxy/zap-core-help/wiki/HelpStartConceptsIntercept [24 de abril de 2019]
[46] https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project#tab=Functionality [24 de abril de 2019]
[47] https://github.com/cloudacademy/static-website-example [12 de mayo de 2019]
[48] https://anchore.freshdesk.com/support/solutions/articles/36000020367-anchore-engine [12 de mayo de 2019]
[49] https://wiki.jenkins.io/display/JENKINS/Anchore+Container+Image+Scanner+Plugin [12 de mayo de 2019]
[50] https://www.redhat.com/es/topics/containers/whats-a-linux-container [13 de mayo de 2019]
[51] https://opensource.com/resources/what-docker [14 de mayo de 2019]
[52] https://openwebinars.net/blog/docker-que-es-sus-principales-caracteristicas/ [14 de mayo de 2019]
[53] https://hub.docker.com/_/httpd [14 de mayo de 2019]
[54] https://github.com/anchore/anchore-engine [14 de mayo de 2019]
[55] https://docs.docker.com/registry/deploying/ [14 de mayo de 2019]
[56] https://docs.docker.com/registry/insecure/ [14 de mayo de 2019]
[57] José Manuel Ortega, DOCKER. Seguridad y monitorización en contenedores e imagen, RC Libros, Madrid, 2019.

[58] https://medium.com/@weekstweets/docker-image-security-for-devsecops-1caf34fc77bc [14 de mayo de 2019]
[59] https://docs.docker.com/engine/reference/builder/ [14 de mayo de 2019]
[60] https://anchore.freshdesk.com/support/solutions/articles/36000020652-policies [14 de mayo de 2019]
[61] https://es.wikipedia.org/wiki/Common_Vulnerabilities_and_Exposures [22 de mayo de 2019]
[62] https://cloudacademy.com/ [22 de mayo de 2019]
[63] https://es.wikipedia.org/wiki/GitHub [22 de mayo de 2019]
[64] https://es.wikipedia.org/wiki/Entorno_de_desarrollo_integrado [22 de mayo de 2019]
[65] https://www.owasp.org/index.php/Main_Page [22 de mayo de 2019]
[66] https://www.welivesecurity.com/la-es/2015/04/21/vulnerabilidad-cross-site-request-forgery-csrf/ [22 de mayo de 2019]
[67] https://es.wikipedia.org/wiki/Bifurcaci%C3%B3n_(desarrollo_de_software) [22 de mayo de 2019]
[68] https://docs.docker.com/registry/introduction/ [22 de mayo de 2019]
[69] https://es.wikipedia.org/wiki/Socket_Unix [22 de mayo de 2019]
[70] https://es.wikipedia.org/wiki/Framework [25 de mayo de 2019]

8. Anexos

8.1 Versiones de las herramientas

COMPONENTE	VERSIÓN
UBUNTU	18.04.2
DOCKER	18.09.3, build 774a1f4
JENKINS	2.164.1
ANCHORE-ENGINE	0.4.0
WARNINGS NEXT GENERATION PLUGIN	4.0.0
OWASP DEPENDENCY-CHECK PLUGIN	4.0.2
OWASP ZAP	2.7.0
HTML PUBLISHER PLUGIN	1.18
ANCHORE CONTAINER IMAGE SCANNER PLUGIN	1.0.18

8.2 Manual instalación Jenkins

Este manual recoge el proceso de instalación y configuración inicial de Jenkins en Docker que permita la ejecución de Dockers (Docker-in-Docker).

Pre-requisito: Tener instalado Docker en Ubuntu [16].

Paso 1: Crear el fichero Dockerfile:

Dockerfile

```
from jenkins/jenkins:lts

USER root

RUN apt-get update -qq \

   && apt-get install -qqy apt-transport-https ca-certificates curl gnupg2 software-properties-common

RUN curl -fsSL https://download.docker.com/linux/debian/gpg | apt-key add -

RUN add-apt-repository \

  "deb [arch=amd64] https://download.docker.com/linux/debian \

  $(lsb_release -cs) \

  stable"

RUN apt-get update  -qq \

   && apt-get install docker-ce docker-ce-cli containerd.io -y

RUN usermod -aG docker jenkins
```

Paso 2: Crear la imagen jenkins-docker utilizando el fichero anterior. Para ello, se ejecutará el siguiente comando en la ruta del fichero:

```
# docker image build -t jenkins-docker .
```

Paso 3: Crear una carpeta que después será compartida con el contenedor:

```
# mkdir /home/$USER/JenkinsData
```

Paso 4: Arrancar el contenedor de la imagen Jenkins-docker creada en el paso anterior.

```
# docker run    -u root   -p 8080:8080   -v \
  /home/$USER/JenkinsData/:/var/jenkins_home   -v \
  /var/run/docker.sock:/var/run/docker.sock   -v "$HOME":/home \
  jenkins-docker:latest
```

Una vez ejecutada esta orden, se creará y arrancará el contenedor. Durante el primer arranque del contenedor, se mostrará una contraseña (Ilustración 22) que deberemos utilizar durante el proceso de configuración de Jenkins.

```
INFO:

*************************************************************
*************************************************************
*************************************************************

Jenkins initial setup is required. An admin user has been created and a password generated.
Please use the following password to proceed to installation:

b9856b0b61c94d83b7b229a5453dfc14

This may also be found at: /var/jenkins_home/secrets/initialAdminPassword

*************************************************************
*************************************************************
*************************************************************
```

Ilustración 22: Contraseña inicial Jenkins

8.3 Manual configuración de proyectos tipo “Pipeline” en Jenkins

Pre-requisitos: Para la correcta ejecución de este proyecto, de deberá haber instalado el Docker de Jenkins tal y como se indica en el anexo 8.2. Y de los plugins correspondientes (Anexo 8.1).

Paso 1: Acceder vía web a Jenkins.

Paso 2: Crear un nuevo proyecto. Para ello, debemos pulsar sobre “New Item” en el menú de Jenkins (Ilustración 23).

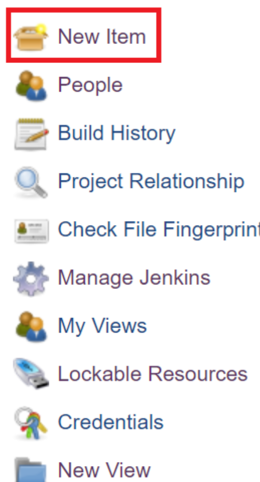


Ilustración 23: Jenkins - New Item

Paso 3: En la siguiente pantalla, introducir un nombre para el proyecto en Jekins y seleccionar el tipo de proyecto “Pipeline” (Ilustración 24).

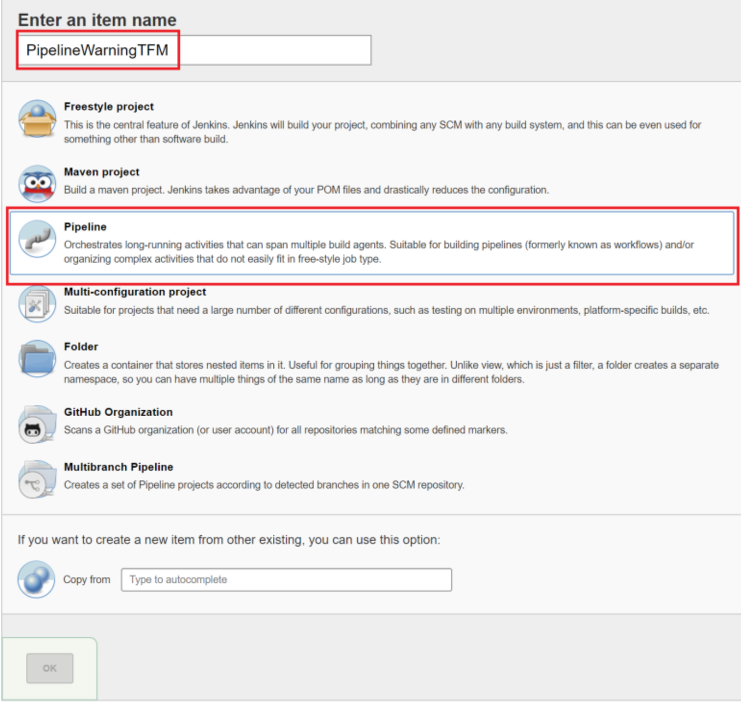


Ilustración 24: Jenkins - Pipeline project

Paso 4: Configurar la sección Pipeline del proyecto con los siguientes parámetros (Ilustración 25):

- Definition: “Pipeline script from SCM”.
- SCM: Git.
- Repository: Según pipeline (Ver Anexos 8.4 y 8.5).
- Credentials: [Nuestras credenciales en GitHub].
- Script Path: Jenkinsfile.
- Desmarcar: Lightweight checkout.

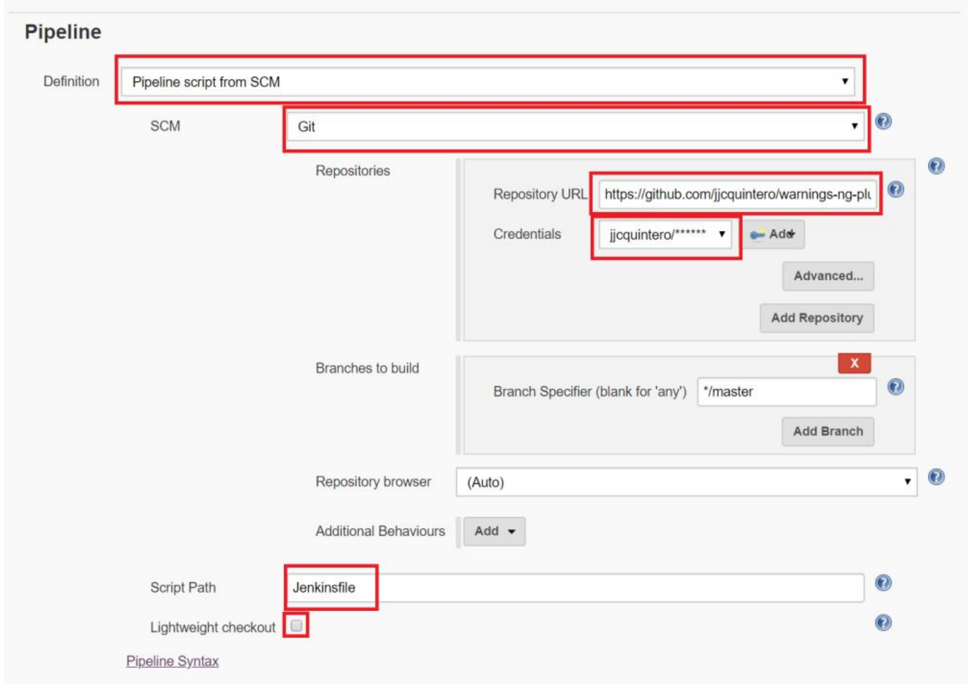


Ilustración 25: Jenkins - Configuración proyecto

8.4 Pipeline control de calidad y dependencias del software

El repositorio de esta pipeline es: https://github.com/jjcquintero/warnings-ng-plugin

Jenkinsfile

```
pipeline {
    agent {
        docker {
            image 'maven:3-alpine'
            args '-v /root/.m2:/root/.m2'
        }
    }
    stages {
        stage('Build') {
            steps {
                sh 'mvn -B -DskipTests clean verify checkstyle:checkstyle pmd:pmd
findbugs:findbugs'
            }
        }

        stage("Dependency Check") {
            steps {
                dependencyCheckAnalyzer datadir: '', hintsFile: '',
includeCsvReports: false, includeHtmlReports: false, includeJsonReports: false,
includeVulnReports: false, isAutoupdateDisabled: false, outdir: '', scanpath: 'src',
skipOnScmChange: false, skipOnUpstreamChange: false, suppressionFile: '',
zipExtensions: ''
            }
        }
    }

    post {
        always {


            recordIssues enabledForFailure: true, tool: mavenConsole(),
referenceJobName: 'Plugins/warnings-ng-plugin/master'
            recordIssues enabledForFailure: true, tools: [java(), javaDoc()],
sourceCodeEncoding: 'UTF-8', referenceJobName: 'Plugins/warnings-ng-plugin/master'
            recordIssues enabledForFailure: true, tool: checkStyle(pattern:
'target/test-classes/io/jenkins/plugins/analysis/warnings/checkstyle.xml'),
sourceCodeEncoding: 'UTF-8', referenceJobName: 'Plugins/warnings-ng-plugin/master'
            recordIssues enabledForFailure: true, tool: cpd(pattern:
'target/cpd.xml'), sourceCodeEncoding: 'UTF-8', referenceJobName: 'Plugins/warnings-
ng-plugin/master'
```

```
            recordIssues enabledForFailure: true, tool: pmdParser(pattern:
'target/test-classes/io/jenkins/plugins/analysis/warnings/recorder/module-
filter/pmd.xml'), sourceCodeEncoding: 'UTF-8', referenceJobName: 'Plugins/warnings-
ng-plugin/master'
            recordIssues enabledForFailure: true, tool: spotBugs(pattern:
'target/test-classes/io/jenkins/plugins/analysis/warnings/spotbugsXml.xml'),
sourceCodeEncoding: 'UTF-8', referenceJobName: 'Plugins/warnings-ng-plugin/master'
            recordIssues enabledForFailure: true, tool:
taskScanner(includePattern:'**/*.java', excludePattern:'target/**/*',
highTags:'FIXME', normalTags:'TODO'), sourceCodeEncoding: 'UTF-8', referenceJobName:
'Plugins/warnings-ng-plugin/master'
            dependencyCheckPublisher canComputeNew: false, defaultEncoding: '',
healthy: '', pattern: '', unHealthy: ''
        }
    }
}
```

8.5 Pipeline pruebas DAST

El repositorio de esta pipeline es: https://github.com/zaproxy/zaproxy

Jenkinsfile

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
              sh "docker rm zapcontainer"
              sh "docker run --name zapcontainer -u root -v $PWD:/zap/wrk -t
owasp/zap2docker-weekly zap-baseline.py -t http://192.168.1.170:8080 -g gen.conf -
a -j -r owaspzap.html || true"
              sh "mkdir -p
$JENKINS_HOME/jobs/$JOB_NAME/builds/$BUILD_NUMBER/htmlreports/OWASP_20ZAP"
              sh "docker cp zapcontainer:/zap/wrk/owaspzap.html
$JENKINS_HOME/jobs/$JOB_NAME/builds/$BUILD_NUMBER/htmlreports/OWASP_20ZAP"
           }
        }
    }
    post {
        always {
           publishHTML target: [
            allowMissing: false,
            alwaysLinkToLastBuild: false,
            keepAll: true,
            reportDir: '$JENKINS_HOME/jobs/$JOB_NAME/builds/$BUILD_NUMBER',
            reportFiles: 'owaspzap.html',
            reportName: 'OWASP ZAP'
          ]
        }
    }
}
```

8.6 Instalación y ejecución de Anchore Engine

<u>Pre-requisitos</u>: Tener instalado docker-compose.

Órdenes para instalar y arrancar Anchore Engine [54]
mkdir ~/aevolume cd ~/aevolume docker pull docker.io/anchore/anchore-engine:latest docker create --name ae docker.io/anchore/anchore-engine:latest docker cp ae:/docker-compose.yaml ~/aevolume/docker-compose.yaml docker rm ae docker-compose pull

Tras ejecutar estas órdenes, se creará el fichero “docker-compose.yaml”. Para arrancar Anchore Engine, se deberá acceder a la ruta donde esté ese fichero para ejecutar el comando:

docker-compose up –d

Una vez ejecutado el comando deberá arrancar Anchore Engine (Ilustración 26).

```
IMAGE                           COMMAND                  PORTS                       NAMES
anchore/anchore-engine:v0.4.0   "/docker-entrypoint.…"   0.0.0.0:8228->8228/tcp      scripts_engine-api_1
anchore/anchore-engine:v0.4.0   "/docker-entrypoint.…"   8228/tcp                    scripts_engine-simpleq_1
anchore/anchore-engine:v0.4.0   "/docker-entrypoint.…"   8228/tcp                    scripts_engine-analyzer_1
anchore/anchore-engine:v0.4.0   "/docker-entrypoint.…"   8228/tcp                    scripts_engine-policy-engine_1
anchore/anchore-engine:v0.4.0   "/docker-entrypoint.…"   8228/tcp                    scripts_engine-catalog_1
postgres:9                      "docker-entrypoint.s…"   5432/tcp                    scripts_anchore-db_1
```

Ilustración 26: Imágenes Anchore Engine

8.7 Manual de configuración de Docker registry

El registry de Docker se ha configurado en una máquina externa a la máquina donde ejecuta Jenkins.

En la máquina externa, se ejecutará el siguiente comando que hará que arranque el registry al inicial la máquina [55]:

```
# docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

Por último, habrá que configurar el host donde se ejecuta Jenkins para permitir el acceso a un registry inseguro. Para ello, bastará con añadir la dirección y el puerto del equipo donde se ejecuta el registry al fichero /etc/docker/daemon.json [56].

/etc/docker/daemon.json

{
"insecure-registries": ["192.168.1.162:5000"]
}

8.8 Manual configuración del proyecto de automatización de la seguridad en la infraestructura en Jenkins

Pre-requisitos: Para la correcta ejecución de este proyecto, de deberá haber instalado el Docker de Jenkins tal y como se indica en el anexo 8.2. Y de los plugins correspondientes (Anexo 8.1). También se deberá haber configurado Anchore Engine como se indica en el anexo 8.6 y el registry de Docker como se indica en el anexo 8.7.

Paso 1: Acceder vía web a Jenkins.

Paso 2: Crear un nuevo proyecto. Para ello, debemos pulsar sobre “New Item” en el menú de Jenkins (Ilustración 23).

Paso 3: En la siguiente pantalla, introducir un nombre para el proyecto en Jekins y seleccionar el tipo de proyecto “Freestyle project” (Ilustración 27).

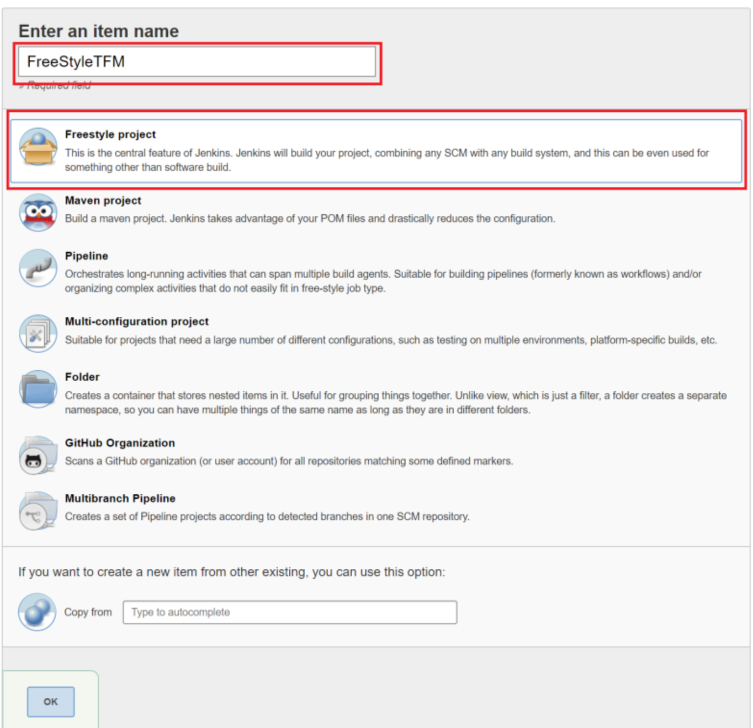


Ilustración 27: Jenkins - Freestyle project

Paso 4: En la sección “Source Code Management”, configurar el repositorio donde está el código fuente de la web [47] (Ilustración 28). En este caso no es necesario añadir ninguna credencial de GitHub, pues se va a actualizar un repositorio público.

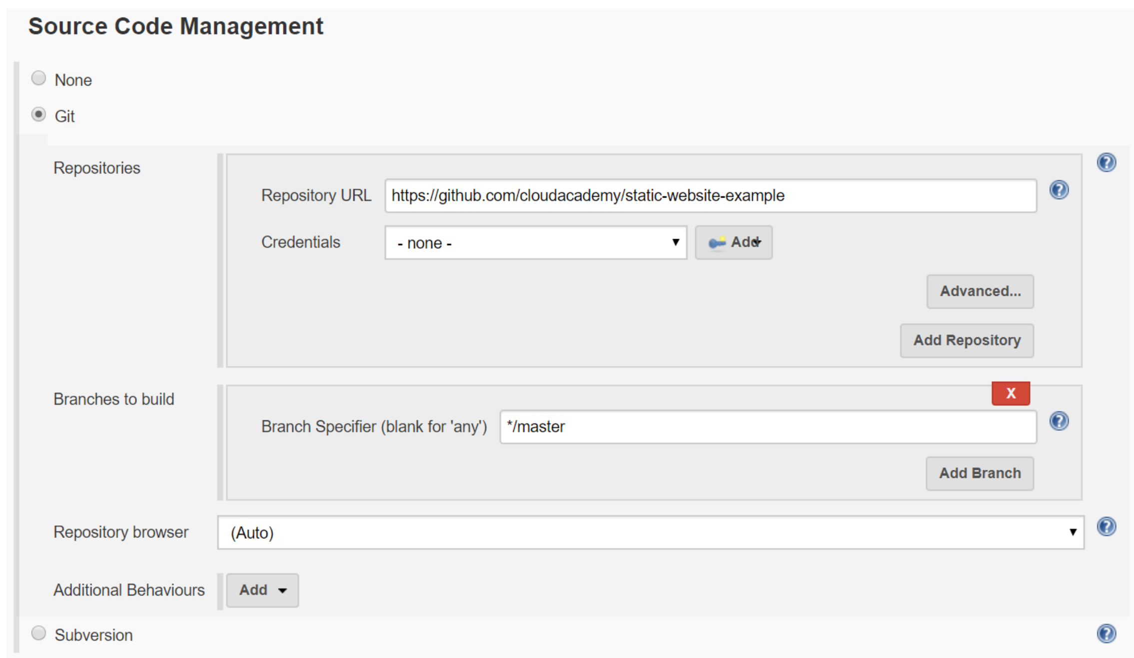


Ilustración 28: Jenkins - Repositorio Web

Paso 5: Añadir el paso "Execute Shell" para poder ejecutar comandos Linux (Ilustración 29). En este caso se han ejecutado los siguientes comandos:

Execute Shell
echo "FROM httpd:latest" > Dockerfile echo "COPY ./static-website-example/ /usr/local/apache2/htdocs/" >> Dockerfile TAG=$(date "+%H%M%S%d%m%Y") REGISTRY=192.168.1.162:5000 IMAGENAME=$REGISTRY/web docker build -t $IMAGENAME:$TAG . docker push $IMAGENAME:$TAG # Line added to create anchore_images file echo "$IMAGENAME:$TAG ${WORKSPACE}/Dockerfile " > anchore_images

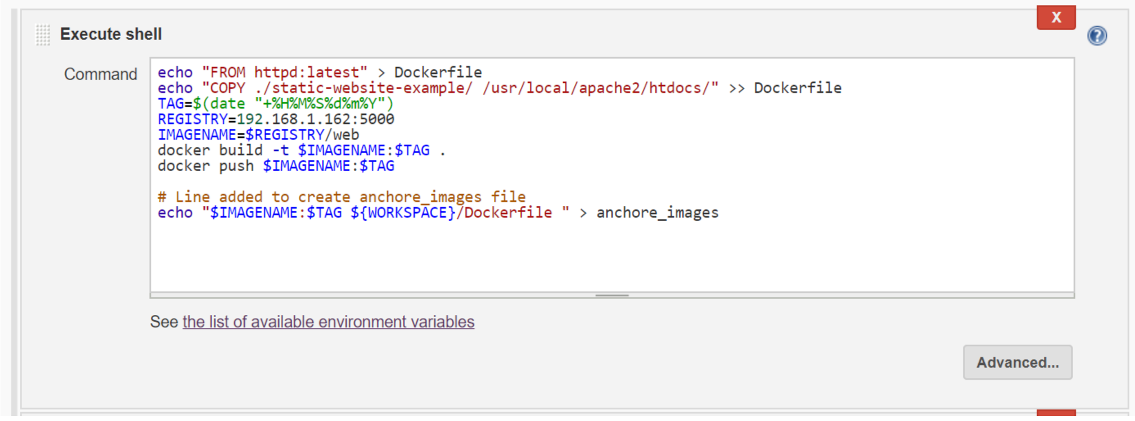


Ilustración 29: Pipeline freestyle - Execute Shell

Paso 6: Añadir el paso "Anchore Container Image Scanner" y establecer su configuración tal que (Ilustración 30):

- Image list file: anchore_images.
- Marcar: Fail build on policy evaluation FAIL result.
- Marcar: Fail build on critical plugin error.
- Anchore Engine operation retries: 300.
- Anchore Engine policy bundle ID: (Vacío).
- Anchore Engine URL: Dirección IP donde corre Anchore Engine.
- Anchore Engine credentials: Usuario y contraseña. Por defecto (admin, foobar).
- Desmarcar: Anchore Engine verify SSL.

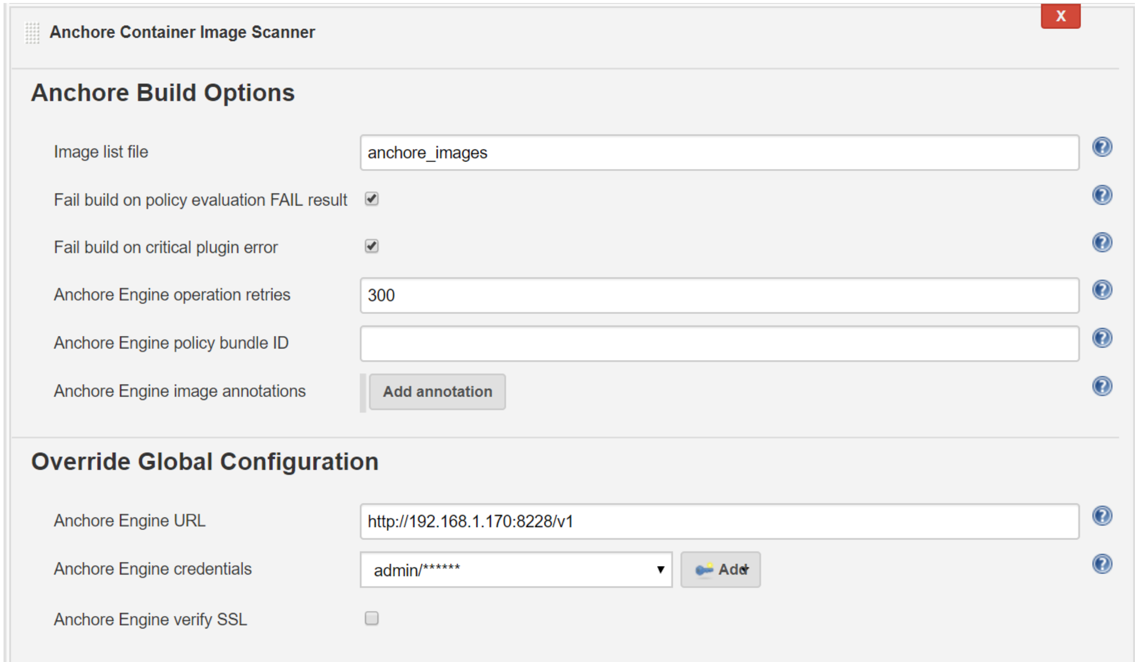


Ilustración 30: Pipeline freestyle - Anchore Engine