Seguridad en el ciclo de vida del desarrollo del software. DevSecOps

Sergio Iriz Ricote

Máster Universitario en Seguridad de las Tecnologías de la Información y de las Comunicaciones

Seguridad Empresarial

Profesor responsable: Pau Del Canto Rodrigo

04/06/2019

QUIERO AGRADECER A MIS PADRES Y A MI HERMANO EL APOYO RECIBIDO DURANTE TODA MI ENSEÑANZA, SIN ELLOS NO HABRÍA SIDO POSIBLE LLEGAR HASTA AQUÍ, EN SEGUNDO LUGAR, MIS AMIGOS Y COMPAÑEROS DE VIAJE QUE SU COMPAÑÍA ME HA HECHO SER MEJOR PERSONA Y APORTARME UNA MAYOR FORTALEZA. A TODOS ELLOS QUIERO DEDICARLES EL PRESENTE PROYECTO.
POR ÚLTIMO, QUIERO AGRADECER A LOS PROFESORES DE LA UOC SU DEDICACIÓN Y ENSEÑANZA EN ESTOS DOS AÑOS DE FORMACIÓN.

Sergio Iriz Ricote

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Seguridad en el ciclo de vida del desarrollo del software. DevSecOps*
Nombre del autor:	*Sergio Iriz Ricote*
Nombre del consultor/a:	*Pau del Canto Rodrigo*
Nombre del PRA:	
Fecha de entrega (mm/aaaa):	06/2019
Titulación:	*Máster Universitario en Seguridad de las Tecnologías de la Información y de las Comunicaciones*
Área del Trabajo Final:	*Seguridad Empresarial*
Idioma del trabajo:	*Castellano*
Palabras clave	*DevSecOps, Jenkins, Gitlab, Clair, Klar, SAST, DAST, SonarQube, ZAP, Owasp, Agile, CI, CD, SDLC, Requisitos, Pipeline*

Resumen Del Trabajo "Seguridad en el Ciclo de Vida del Desarrollo del Software. DevSecOps"

En la actualidad debido a la gran demanda de productos, las empresas necesitan realizar desarrollos más rápidos y disponer de entregables en menor tiempo, por ello emplean metodologías ágiles en su ciclo de vida del desarrollo. Esto tiene innumerables ventajas, pero en muchas ocasiones se descuida la seguridad y calidad de los entregables, además la cultura de las empresas tiene la conciencia de que la seguridad es un paso al final del SDLC, limita las entregas y es cuestión de los expertos en seguridad.

El objetivo de este trabajo de fin de máster es cambiar esa idea y demostrar la importancia de la seguridad, en todas las fases del ciclo de vida del software, de una manera ágil e integrada en el SDLC. Esto ha sido posible gracias a DevSecOps que integra tanto Operaciones, como Desarrollo y Seguridad.

Por ello se ha diseñado e implementado un ciclo de vida DevSecOps, que incluye pruebas de seguridad automáticas e integradas en el ciclo de vida del desarrollo del software en todas sus fases, que nos permite conocer en todo momento el estado de seguridad de la aplicación y sus riesgos de manera incremental en un sistema de despliegue continuo.

Con ello se ha demostrado que la seguridad involucra a todos los equipos y fases que mantienen y desarrollan una aplicación durante el SDLC, y se ha eliminado la idea de que es costosa y limitante en tiempo de entrega. Siendo imprescindible unir Seguridad, Desarrollo y Operaciones.

Abstract: Security in the Software Development Life Cycle. DevSecOps

Currently, due to the high demand for products, companies need to develop faster and have deliverables in less time, so they use agile methodologies in their development life cycle. This has innumerable advantages, but in many cases the safety and quality of the deliverables is forgotten, besides, the culture of the companies is aware that safety is a step at the end of the SDLC, it limits the deliveries and it is a matter of the experts in security.

The objective of this end of master project is change that idea and demonstrate the importance of security, in all the software life cycle phases, in an agile and integrated way in the SDLC. This has been possible thanks to DevSecOps that integrates both Operations, Development and Security.

For this reason, a DevSecOps life cycle has been designed and implemented, which includes automatic security tests integrated into the life cycle of software development in all its phases, which allows us to know the security status of the application at all times and their risks incrementally in a continuous deployment system.

This has shown that security involves all the equipment and phases that maintain and develop an application during the SDLC, and the idea that it is expensive and limiting at delivery time has been eliminated. Being essential to unite Security, Development and Operations.

Tabla de Contenidos

ÍNDICE DE TABLAS

ÍNDICE DE ILUSTRACIONES

1. INTRODUCCIÓN

1.1 CONTEXTO

La metodología de desarrollo del Software se ha enmarcado en diferentes cambios, respondiendo a las necesidades de cada momento y el conocimiento de las diferentes técnicas de desarrollo. Para ello, la Ingeniería del software es la ciencia que se ha encargado de definir los procesos cuya finalidad es desarrollar productos o soluciones para un cliente o mercado en particular, teniendo en cuenta factores como los costes, la planificación, la calidad y las dificultades asociadas. A todo esto, es a lo que denominamos metodologías de desarrollo de software (School, s.f.) es decir, trata los diferentes procesos que deben seguirse a la hora del diseño, implementación y despliegue de una solución.

1.1.1 METODOLOGÍAS DE DESARROLLO HASTA LA ACTUALIDAD. SDLC, DEVOPS & ALM

La sociedad está en constante cambio, la tecnología y las necesidades de las personas, empresas y clientes evoluciona a un ritmo vertiginoso. Estos cambios afectan a todos los ámbitos de la vida y la gestión de proyectos no iba a ser menos.

El desarrollo de software en sus inicios era artesanal en su totalidad, la fuerte necesidad de mejorar el proceso y llevar los proyectos a la meta deseada, tuvieron que importarse la concepción de metodologías existentes en otras áreas y adaptarlas al desarrollo de software. Esta nueva etapa de adaptación contenía el desarrollo dividido en etapas de manera secuencial que de algo mejoraba la necesidad latente en el campo del software. (Roberth G. Figueroa1, 2007)

Estas metodologías eran muy poco flexibles y con tiempos de desarrollo muy amplios, los costes para implementar cambios son muy altos, no ofrecen una buena solución en entornos volátiles y las etapas se definen una vez, y hasta no finalizar una etapa no se ejecuta la siguiente. Por último y de gran importancia, el cliente no ve el producto hasta su finalización, además es inviable incorporar la seguridad hasta la última etapa del desarrollo del software. Un ejemplo de estas metodologías el ciclo de vida del software en cascada:

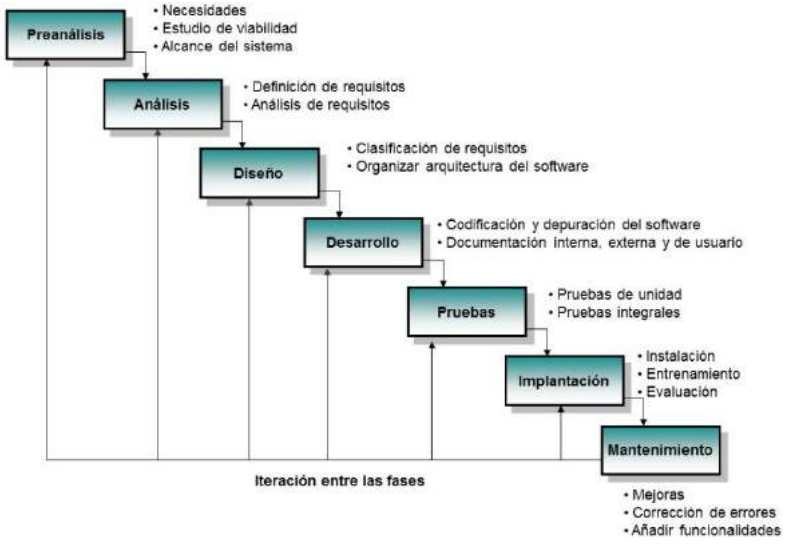

Ilustración 1: Metodología en cascada

Como se observa en el diagrama, se trata de un enfoque metodológico que ordena las etapas del proceso del desarrollo del software, de tal forma que el inicio de cada etapa debe esperar al final de la anterior.

Por todos estos motivos, y este entorno cambiante y evolutivo constante, surgen las metodologías ágiles y se basa en dos aspectos puntuales, el retrasar las decisiones y la planificación adaptativa; permitiendo potencia aún más el desarrollo de software a gran escala.

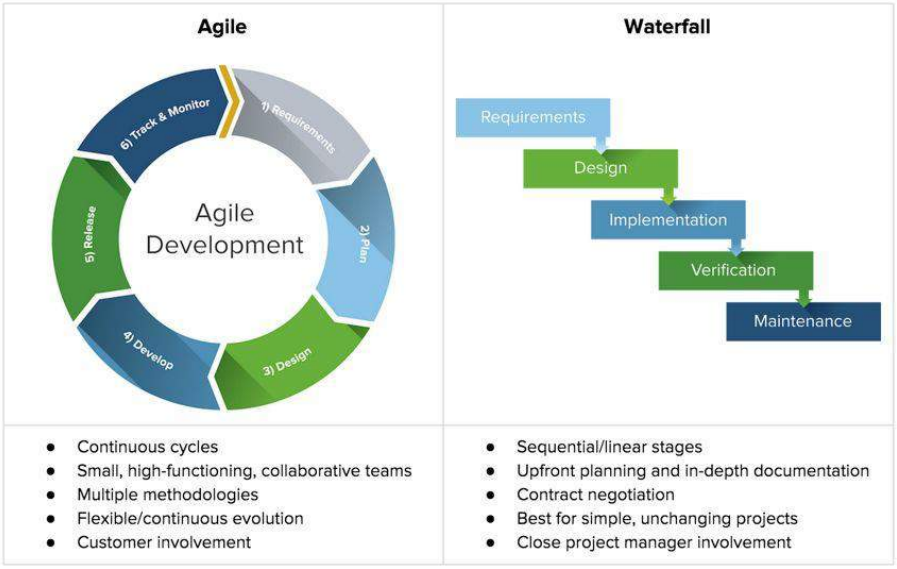

Ilustración 2: Agile VS Cascada

Las características principales y que diferencia Agile de las metodologías tradicionales como Cascada, es que se trata de un proceso iterativo, que incluye al cliente en todas sus distintas fases y favorece la interacción entre los diferentes elementos.

Sin embargo, todas estas metodologías hacen difícil la comunicación entre los diferentes equipos de operaciones y los desarrolladores. Por ello, nace DevOps una evolución de las metodologías ágiles, sus objetivos primordiales es conseguir la integración perfecta entre desarrolladores y administradores de sistemas y abandonar los flujos inoperativos y revisiones continuas del trabajo de unos y otros, para empezar a trabajar de forma unidireccional y colaborativa. La fluidez de su sistema de trabajo permite fabricar software de gran calidad en el menor tiempo posible y con el coste justo. Sin embargo, no solo cambia la forma de desarrollar, sino que también implica un cambio cultural.

Hay varios aspectos en los que difieren:

Devops trae las prácticas del método Agile a la administración de sistemas. Como la entrega por plazos y la apuesta por el desarrollo en equipo, pero hay ciertos aspectos en los que difieren, entre ellos: Devops se centra en la automatización y Agile no tanto, porque, aunque lo considera útil, también lo ve como un foco de distracciones. Es un método de trabajo muy dirigido a los resultados que extiende el proceso y lo completa mediante la integración continua de nuevas funcionalidades y actualizaciones. Asegura el lanzamiento en plazos razonables, permitiendo que el código esté listo para producción y proveyendo de valor al cliente. (Avansis, 2018)

DevOps está ayudando a los equipos de desarrollo y operaciones a trabajar juntos de manera más eficaz al fomentar una mejor comunicación y colaboración. Mover la creación, el paquete y la implementación de la aplicación en forma ascendente permite que las operaciones automaticen todo el proceso de implementación en una etapa más temprana del ciclo de vida y también obtengan los conocimientos necesarios para comprender cómo admitir la aplicación.

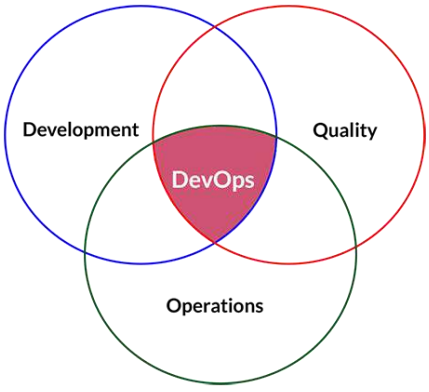

Ilustración 3: DevOps

Pero ¿cómo consigue la comunicación y colaboración esto DevOps? Para entender esto hay que conocer el concepto de "Application Lifecycle Management", conocido como ALM o Gestión de ciclo de vida de aplicaciones, en el que está inmerso, se refiere al proceso de incorporar, coordinar y monitorear todas las actividades necesarias para la creación de una solución de software o aplicación, desde el nacimiento de la necesidad, pasando por su definición, su desarrollo, su despliegue y su posterior mantenimiento" (consultor-it, 2016)

ALM tiene sus raíces en el ciclo de vida del desarrollo de software (SDLC), con etapas típicamente definidas que incluyen requisitos, diseño, desarrollo y pruebas. Sin embargo, el ALM en realidad tiene una visión mucho más amplia.

El ALM define las tareas, roles y responsabilidades que se requieren para soportar todo el ciclo de vida del software y los sistemas. Esto incluye la definición de requisitos, diseño, desarrollo, pruebas, pero también otras funciones relacionadas, incluidas las funciones de soporte de sistemas en curso. ALM adopta un enfoque integral e integrado y se basa en una herramienta robusta de automatización del flujo de trabajo para su organización y éxito. El ALM proporciona transparencia, trazabilidad y, lo más importante, la gestión del conocimiento y la comunicación, que es la lección clave que aprendemos de los DevOps efectivos. Si bien los equipos multifuncionales suelen ser ideales y muy efectivos, a veces las organizaciones deben mantener una separación de controles. (Aiello, 2013)

1.1.2 CICLO DE VIDA DEL DESARROLLO DEL SOFTWARE

Todas estas metodologías de desarrollo tienen como objetivo definir un plan de acción para el desarrollo del software, desde el momento en que comienza a planificar un proyecto comienza el Ciclo de Vida del Desarrollo del Software en inglés SDLC (Software Development Life Cycle) como me referiré a partir de ahora, siendo "todo el proceso que tiene cualquier desarrollo nuevo en una organización desde que se tiene la idea del mismo hasta que está ya implantado en producción" (Saavedra, 2018). A continuación, identifico un SDLC Ágil, que engloba las distintas fases:

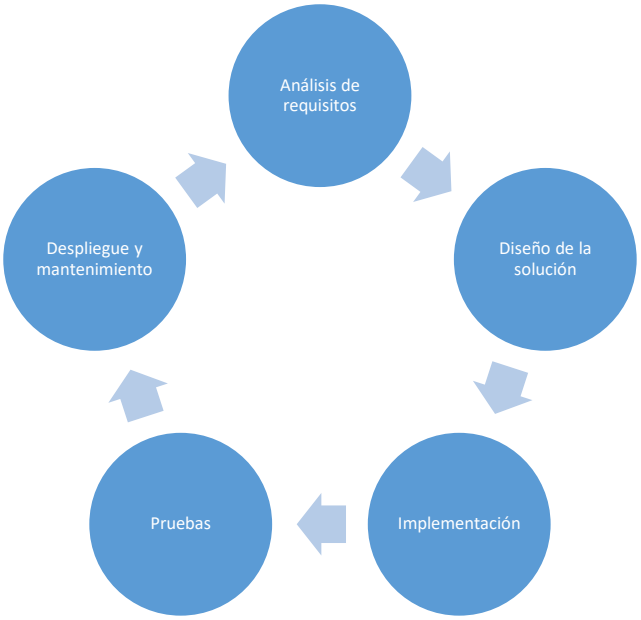

Ilustración 4: Ciclo de Vida Del Desarrollo del Software

- Planificación. Análisis y requisitos: El SDLC comienza con la fase de análisis de requisitos, donde los interesados discuten los requisitos del software que debe desarrollarse para lograr un objetivo. El objetivo de la fase de análisis de requisitos es capturar el detalle de cada requisito y asegurarse de que todos comprendan el alcance del trabajo y cómo se va a cumplir cada requisito.

- Diseño. La siguiente etapa del ciclo de vida del desarrollo de software es la fase de diseño. Durante la fase de diseño, los desarrolladores y arquitectos técnicos inician el diseño de alto nivel del software y el sistema para poder cumplir con cada requisito.

- Desarrollo. Se implementa la solución de acorde al diseño y los requisitos.

- Pruebas. Es la última fase del ciclo de vida del desarrollo del software antes de que el software se entregue a los clientes. Durante las pruebas, los probadores experimentados comienzan a probar el sistema contra los requisitos.

- Despliegue y mantenimiento. Una vez que el software se haya probado por completo y no haya problemas, es hora de implementarlo en la producción donde los clientes pueden usar el sistema. (Ghahrai, 2018)

1.1.3 DEVOPS

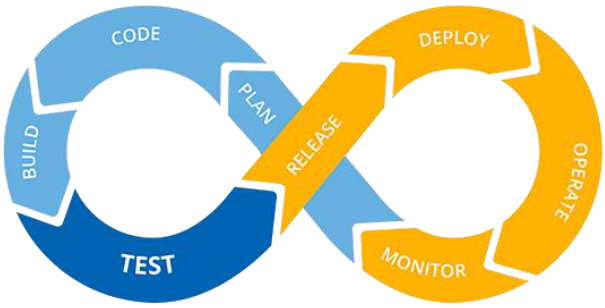

Ilustración 5: Ciclo DevOps

DevOps mantiene un SDLC iterativo El proyecto es planificado en base a distintos bloques que se repiten en cada interacción, produciéndose así unos beneficios de manera creciente que permitirán hacer entrega de un producto final más completo.

- Planificación. Aquí se planifica y programa la realización de las tareas asincrónicas, como la definición de los requisitos de la solución.
- Desarrollo e integración continua. En esta fase se diseña la arquitectura, los testers definen las pruebas y se realiza la implementación del código por parte de los desarrolladores. En esta fase se automatizan los mecanismos de revisión, validación, pruebas y alertas construidos en las iteraciones, permitiendo identificar rápidamente posibles fallos (es decir, en qué pieza y línea de código está el error). Este proceso sucede cada vez que se actualiza el código del repositorio.
- Despliegue. Lo que DevOps promueve en esta fase es la automatización de los despliegues por medio de herramientas y scripts que permiten resumir la validación de todo proceso en un botón de aprobación.
- Operaciones. En esta fase se obtienen métricas necesarias para revisar el funcionamiento del software, permitiendo modificar de forma dinámica la infraestructura para garantizar la escalabilidad, persistencia, disponibilidad, transformación y resiliencia.
- Monitorización. En esta fase es donde se definen los requisitos y condiciones que monitorizaremos para controlar la salud de las aplicaciones y su infraestructura. (Bienvenido, 2018)

1.1.4 PROBLEMAS DE SEGURIDAD EN LAS ETAPAS DEL CICLO DE VIDA DEL DESARROLLO

A pesar de los avances en las metodologías de desarrollo, la seguridad, siempre es una asignatura pendiente, en los flujos de trabajo tradicionales, la seguridad se aplica al final del SDLC o al encontrarse el software en entornos productivos provocando un aumento significativo de los plazos de entrega y un aumento considerable en el coste de corrección de las vulnerabilidades encontradas, debido a la complejidad de implementar una solución en un entorno ya desplegado tanto en la búsqueda de la vulnerabilidad como su corrección.

Además, en los ciclos de desarrollo ágiles, se prioriza la rapidez y el número de entregables en pequeños periodos de tiempo, descuidando la calidad y la seguridad de las releases.

- Planificación. En la fase de análisis y requisitos se centran en el cumplimiento regulatorio de calidad y la definición de requisitos relacionados con la funcionalidad y el objetivo de la solución, descuidándose una definición de políticas de seguridad que deban cumplirse, como patrones de desarrollo y despliegue seguros.

- Diseño. En la fase de diseño, no se tienen en cuenta los patrones de diseño seguro.

- Desarrollo. En las distintas metodologías, en la fase de desarrollo el objetivo principal es el cumplimiento de requisitos y objetivos de funcionalidad, además los desarrolladores piensan que la seguridad tiene como responsable a expertos analistas, descuidándose así una de las fases más importantes de la seguridad. Se incorporan, librerías de terceros cuyo código fuente no es accesible y en numerosas ocasiones contienen vulnerabilidades críticas, que pueden comprometer la seguridad de la aplicación.

- Pruebas. Las pruebas se orientan a la integración y calidad QA de los entregables, siendo esencial incorporar pruebas de seguridad que garanticen los requisitos de seguridad.

- Despliegue. Se realizan despliegues en entornos inestables e inseguros, la configuración de los entornos no considera los parámetros de seguridad. No se realizan pruebas de seguridad sobre entornos productivos. Todo esto produce que, si no se ha realizado un despliegue sin el correcto bastionado y configuraciones de seguridad, un usuario malicioso pueda penetrar en el sistema y vulnerarlo, suponiendo incluso un gran riesgo para las organizaciones.

1.2 MOTIVACIÓN DEL TRABAJO DE FIN DE MÁSTER

1.2.1 NECESIDAD DE INCORPORAR SEGURIDAD DE FORMA ÁGIL

Como se mencionó en los anteriores puntos, las metodologías de desarrollo han ido evolucionando en paralelo a las necesidades de los clientes y las organizaciones, de esa misma manera en los últimos años el número de ataques y vulnerabilidades de Zero Day descubiertas se han ido multiplicando.

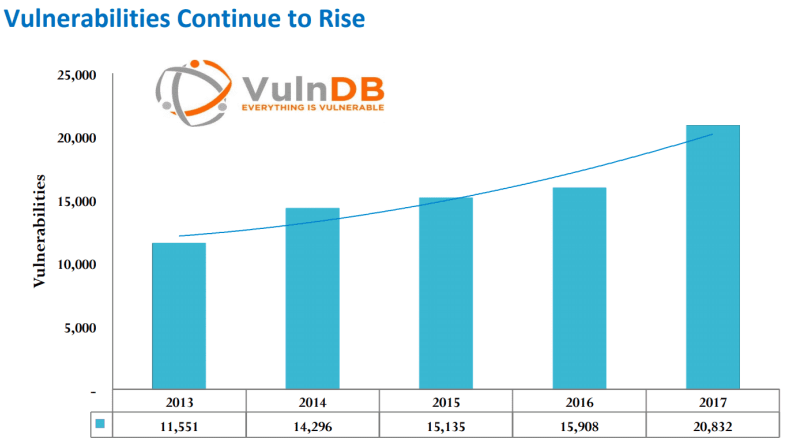

Ilustración 6: Crecimiento del número de vulnerabilidades por año

Por ello debe producirse un feedback continuo de la seguridad en todas las etapas, y eliminar la cultura de la seguridad al final del ciclo como única herramienta de seguridad.

La seguridad al final del ciclo aumenta los tiempos de entrega y reduce la flexibilidad.

Los tiempos de remediación según empresas como Checkmarx, Veracode y Black Hat de las vulnerabilidades en las fases más tempranas del ciclo de vida es mucho menor, y por ende el coste de remediación y la complejidad.

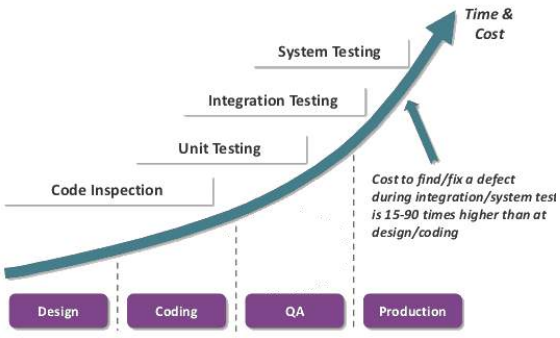

Ilustración 7: Checkmarx, coste de remediación por fase

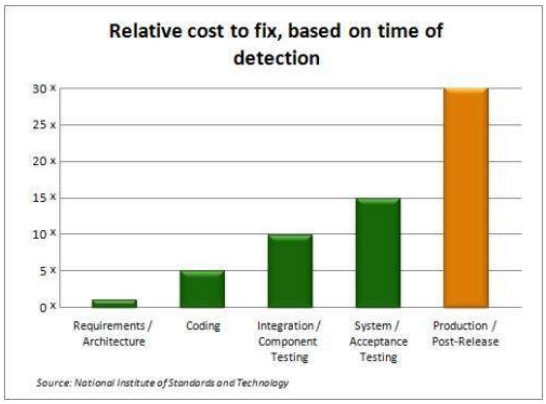

Ilustración 8: Veracode, coste de remediación por fase

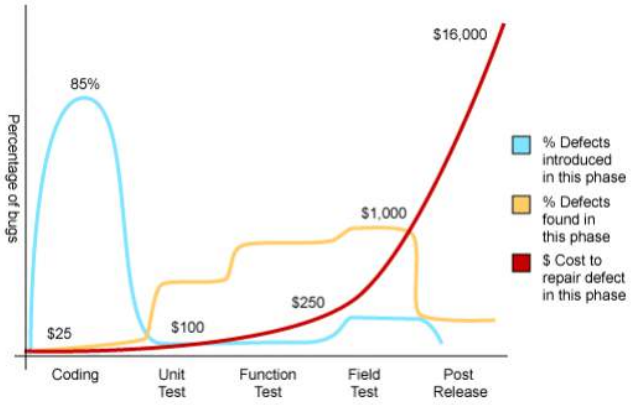

Ilustración 9: BlackHat, coste de remediación por fase

Esto propicia una necesidad imperativa de idear un sistema que permita la integración de la seguridad desde las primeras fases del SDLC hasta el final de ciclo, y debe realizarse de forma continua e iterativa.

Se trata de una necesidad y una oportunidad de negocio que debe ofrecerse a las diferentes empresas, comenzando por cambiar la cultura y concienciando de que la seguridad es un compañero de viaje del SDLC.

DevOps cuya filosofía principal es eliminar los puentes entre los diferentes equipos que intervienen en el SDLC y en las operaciones produciendo un cambio de cultura en las empresas, es la herramienta perfecta para incorporar seguridad, y nuevamente eliminar las barreras con los desarrolladores, las operaciones y la seguridad:

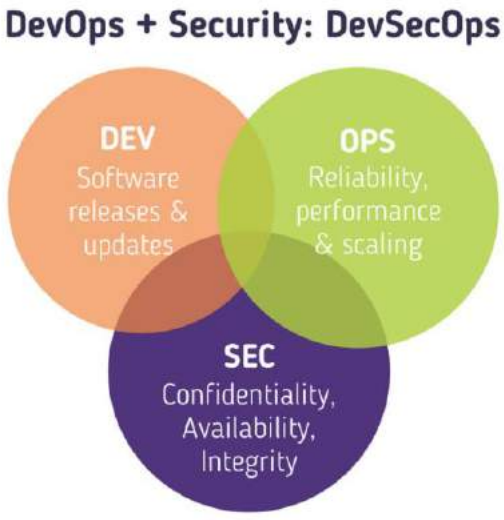

Ilustración 10: DevSecOps

1.3 OBJETIVOS

1.3.1 OBJETIVOS GENERALES

El objetivo principal es la definición e implantación de un ciclo de vida del desarrollo del software seguro DevSecOps.

Además, se quiere definir una serie de políticas y requisitos de seguridad, como la definición e implantación de un sistema de despliegue continuo, incluyendo de forma automática la seguridad en cada una de las fases y siendo integrable en los distintos pipelines DevOps.

- Diseño de un ciclo de vida del desarrollo del software seguro
- Análisis y definición de políticas y requisitos de seguridad que debe cumplir el software.
- Implementación de un sistema de despliegue continuo.
- Pruebas de seguridad en cada una de las fases del ciclo de vida del software.
- Integración de las políticas de seguridad y el sistema de despliegue continuo en el ciclo de vida DevSecOps.

1.4 METODOLOGÍA

Para cumplir los objetivos prefijados, como se ha comentado en el marco de trabajo, se enmarcará en la cultura DevOps y las diferentes herramientas y metodologías que dispone.

1.4.1 DEVOPS

La cultura DevOps se centra en los diferentes equipos involucrados en el desarrollo del software. Para ello se debe centrar en los cuatro factores, que en ocasiones pueden tener problemas de comunicación, sinergias o encontrarse en fases muy distintas de trabajo.

Desarrolladores: Centrados en la creación e implementación de sistemas informáticos basados en lenguajes de programación.

Operaciones: Personas encargadas de procesos de producción, desarrollo de hardware, etc.

Control de calidad: Un tercer grupo que se encarga de testar productos y tomar decisiones.

Seguridad. Analistas y auditores que se encargan de dotar al sistema de una seguridad y robustez frente a las diferentes amenazas exteriores o interiores al software.

Devops debe conciliar el trabajo de estos tres departamentos en busca de la consecución de objetivos comunes.

Por este motivo, empleando la cultura y principios Devops, el desarrollo de este trabajo se apoyará en la premisa de eliminar las barreras entre los distintos equipos de trabajo, el desarrollo y la seguridad, incluyendo la **seguridad en el proceso de integración continua, de una forma automatizada y ágil**, permitiendo optimizar el proceso de desarrollo y de auditoría de seguridad para poder reducir los tiempos de paso a producción y de entrega al cliente, así como una **definición conjunta de requisitos del sistema, de la empresa y del software sin perder el foco de la seguridad**, haciendo visible para todos los equipos que utilizan esta especificación los distintos puntos de vista.

1.4.2 SEGURIDAD SHIFTLEFT

Para evitar el retraso en la entrega de los proyectos, debe tenerse en cuenta la seguridad desde la fase de diseño. Para conciliar agilidad y seguridad, la solución reside en **implementar la seguridad desde el inicio del proyecto**, no una vez se ha finalizado.

La integración de la seguridad en un ciclo de desarrollo ágil debe empezar lo más pronto posible, esto es, en la fase de definición de requisitos. Este enfoque, se llama Seguridad Shift Left, es decir **orientación hacia el principio del enfoque de la seguridad**, que permite en el proceso de desarrollo del software tener un flujo de trabajo totalmente seguro en cada etapa del ciclo de desarrollo del proyecto.

Para ello, hay que incluir la integración de la seguridad en los procesos operativos y de desarrollo mediante la implementación de sistemas y procesos automáticos que no solo sean capaces de detectar y alertar de los problemas de seguridad, sino también de reaccionar en caso de incidencia.

La implementación de estos mecanismos permite identificar y tratar todos los problemas de seguridad. No deben considerarse como frenos a la innovación, puesto que eliminan los cuellos de botella y permiten entregar más rápido soluciones seguras.

1.4.3 OPEN SOURCE

La solución que se va a plantear se basa en las premisas de una implementación de bajo coste e integrable en cualquier proceso de desarrollo DevOps empleando **herramientas y especificaciones Open Source**, que no supongan una limitación para integrarse en cualquier proceso ya sea por coste o por política de la organización.

1.5 PLANIFICACIÓN

Seguridad en el ciclo de vida del software. DevSecOps

Encargado del proyecto	Sergio Iriz Ricote
Fechas de inicio y fin del proyecto	20-feb-2019 - 18-jun-2019
Progreso	100%
Tarea	17
Recursos	1

Nombre	Fecha de inicio	Fecha de fin
Entrega 1	20/02/19	5/03/19
Introducción y objetivos	20/02/19	5/03/19
En esta fase se definirán los objetivos del proyecto y el contexto en el que se enmarca. Además, se expondrá el problema a resolver y la planificación del mismo.		
Entrega 2	6/03/19	2/04/19
Definición del marco teórico del proyecto	6/03/19	11/03/19
En esta fase se realizará la investigación y definición del marco teórico del problema. Definiciones y conceptos necesarios para la comprensión del problema abordado		
Diseño del ciclo de vida seguro del software. DevOps	8/03/19	20/03/19
Fase de diseño del Ciclo de Vida del Desarrollo del Software Seguro. En esta fase se realizará un diseño de un SDLC incorporando la seguridad.		
Selección de herramientas para el S-SDLC	20/03/19	21/03/19
En esta fase se seleccionarán las herramientas que se utilizarán para la implementación de la solución.		
Definición de fase de planificación segura. Definición de requisitos y políticas de seguridad	22/03/19	2/04/19
Fase de implementación de la seguridad en la fase de Planificación del S-SDLC. Se definirán los requisitos que debe cumplir un proyecto y las políticas de seguridad que deben aplicarle.		
Entrega 3	2/04/19	30/04/19
Implementación de la integración continua de la seguridad	2/04/19	20/04/19
Integración de la seguridad en la fase de integración continua del desarrollo. en esta fase se diseñará e implementará el sistema de integración continuo de la seguridad.		
Integración continua SAST	2/04/19	13/04/19
En esta fase se implementará el ciclo de integración continua de análisis estático de código		
Integración continua. Análisis de dependencias	15/04/19	20/04/19
En esta fase se implementará el ciclo de integración continua del análisis de dependencias.		
Integración en el S-SDLC	20/04/19	30/04/19
En esta fase, se realizará la integración de la seguridad en un pipeline DevOps		
Documento Final	20/02/19	4/06/19
Documento Final	20/02/19	4/06/19
Elaboración y redacción del informe final del documento del trabajo de fin de máster		

Nombre	Fecha de inicio	Fecha de fin
Vídeo presentación	6/06/19	11/06/19
Vídeo presentación	6/06/19	11/06/19
Elaboración y preparación del vídeo de defensa del trabajo		
Preparación de la defensa	12/06/19	17/06/19

Diagrama de Gantt

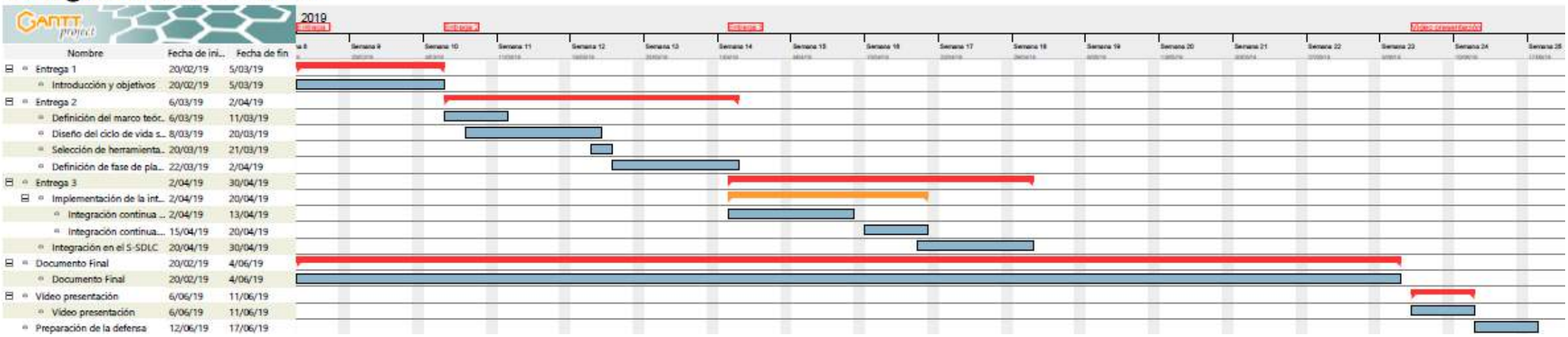

Ilustración 11: Gannt

2. PLANTEAMIENTO DEL PROBLEMA

Un reciente informe de CA Veracode sobre el estado de la seguridad del software resaltaba que "las vulnerabilidades continúan apareciendo a un ritmo alarmante en software que no han sido testado previamente desde el punto de vista de la seguridad y que organizaciones de todo el mundo reconocen que el 77% de las aplicaciones tiene al menos una vulnerabilidad en un análisis inicial." (Veracode, 2018) Cierto es que con el paso del tiempo las empresas son cada día más conscientes de que desarrollando de forma segura es una clave muy importante para su evolución hacia modelos digitales.

No obstante, a pesar de este tipo de indicadores, la premisa de integrar la seguridad en todas las fases del desarrollo no se cumple. El gran problema y motivo por el que sucede esto, es la gran dificultad por parte de las empresas de integrar prácticas de seguridad como parte del ciclo de vida del desarrollo del software.

En su mayoría, el factor común por el que la seguridad no se considera un objetivo del SDLC, es la consideración de que es una comprobación que se realiza al final del ciclo de vida una vez la aplicación ya se encuentra en entornos productivos, además, no hay tiempo para incluir la seguridad porque limita los tiempos de entrega ya que la mayoría de los desarrollos van muy ajustados de tiempo y es más prioritario realizar la entrega en el plazo adecuado que incluir la seguridad en cada una de sus partes. Por último, la visión de que la seguridad es trabajo de los especialistas en seguridad, y que los desarrolladores y las operaciones no deben tener consciencia de ello por este motivo.

Toda esta consciencia negativa de la seguridad se debe principalmente a las limitaciones de las metodologías de desarrollo existentes.

Las metodologías tradicionales de desarrollo con Cascada, que derivan la seguridad al final del ciclo, no se adapta a la idea de entregas ágiles y sin retrasos, posicionando la seguridad como un paso bloqueante y no como un apoyo en el desarrollo.

En las metodologías de desarrollo ágiles existen unos grupos claramente diferenciados, entre desarrolladores y operaciones, por este hecho la seguridad no tiene consciencia de las necesidades de los desarrolladores, y los desarrolladores no son partícipes de la seguridad, provocando una barrera cultural dentro de las organizaciones.

El objetivo prioritario de este trabajo es eliminar este tipo de consciencias y pensamiento acerca de la seguridad como parte fundamental del ciclo de vida del software. Para ello planteando la cultura y metodologías DevOps se va a integrar la seguridad como un agente más del SDLC, incluyendo a las operaciones y a los desarrolladores en la consciencia de la seguridad, siendo partícipes de forma activa y colaborativa, eliminando la consciencia de que la seguridad es algo bloqueante, que produce retrasos en las entregas, y que es algo intrínseco a las fases finales.

Para lograrlo, se debe definir un ciclo de vida del desarrollo del software seguro, enmarcando la seguridad en cada una de las etapas, empleando:

- Metodologías Ágiles y DevSecOps
- Definición de requisitos y políticas de seguridad para la fase de diseño y planificación del proyecto
- Integración de seguridad en la fase de integración continua. Definición de un pipeline seguro.

3. MARCO TEÓRICO

Para la resolución del problema planteado hay que tener en cuenta una serie de conceptos y definiciones que se desarrollaran en la solución planteada:

3.1 DEVSECOPS

El cambio hacia el aprovisionamiento dinámico, los recursos compartidos y la computación en la nube han generado beneficios en torno a la velocidad, agilidad y costo de IT, y todo esto ha ayudado a mejorar el desarrollo de aplicaciones. La capacidad de implementar aplicaciones en la nube ha mejorado tanto la escala como la velocidad, el cambio a las metodologías ágil y DevOps especialmente con el principio de integrar el desarrollo y las operaciones como un conjunto único y basado en la automatización, ha ayudado con todo, desde lanzamientos de funciones más frecuentes hasta una mayor estabilidad de la aplicación.

Sin embargo, muchas herramientas de seguridad y cumplimiento no se han mantenido al día con este ritmo de cambio, ya que simplemente no fueron diseñadas para probar el código a la velocidad que DevOps requiere. Otras técnicas de auditoría como el hacking son procesos lentos que paralizan los desarrollos. Esto solo ha consolidado la opinión de que la seguridad es el mayor bloque para el rápido desarrollo de aplicaciones y, más en general, la innovación de IT.

DevSecOps se trata de la unión de la cultura DevOps y la seguridad. Trata sobre como introducir seguridad desde las primeras fases de ciclo de vida del desarrollo del software, minimizando así las vulnerabilidades y acercando la seguridad a los objetivos de TI y de negocio.

La premisa principal de DevSecOps es que todo agente involucrado en el ciclo de vida del software es responsable de la seguridad, en esencia es la colaboración entre desarrolladores, operaciones con funciones de seguridad, incluyendo seguridad en todas las fases del desarrollo del software.

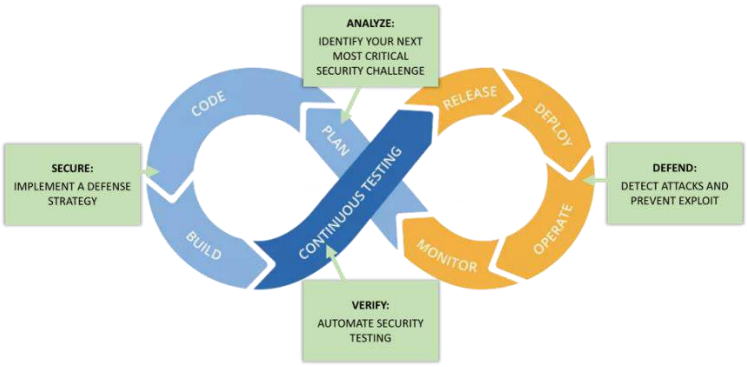

Ilustración 12: Ciclo de vida DevOps

Para ello uno de los pilares fundamentales de esta cultura es la automatización de las tareas principales de seguridad, incluyendo controles de seguridad en el flujo de trabajo DevOps.

Es el caso de la inclusión de seguridad en los procesos de integración continua, la planificación del desarrollo teniendo en cuenta requisitos de seguridad, el despliegue de la nueva release en entornos securizados previamente a partir de pruebas de seguridad, o en el caso de emplear contenedores como Docker, el análisis de las imágenes Docker. (Drinkwater, 2019)

Los beneficios principales de esta nueva cultura de operaciones, desarrollo y seguridad son claros:

- Una mayor automatización desde el principio reduce la posibilidad de una mala administración y reduce los errores.
- Según Gartner, “DevSecOps puede llevar a que las funciones de seguridad como la administración de identidades y acceso (IAM), el firewall y el análisis de vulnerabilidades se habiliten mediante programación a lo largo del ciclo de vida de DevOps, lo que deja a los equipos de seguridad libres para establecer políticas”
- Todos los equipos tienen conciencia de seguridad. Se reducen incidentes y mejora la seguridad a través de la responsabilidad compartida.
- Se reducen los costes de reparación de vulnerabilidades.

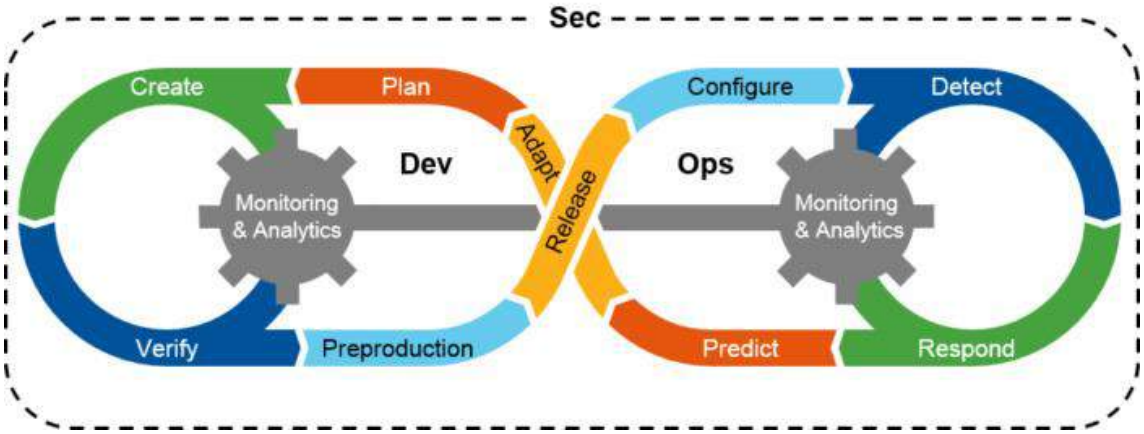

Ilustración 13: DevSecOps

3.2 ANÁLISIS ESTÁTICO DE LA SEGURIDAD DEL CÓDIGO (SAST)

Según Gartner "Static Application Security Testing (SAST) es un conjunto de tecnologías diseñadas para analizar el Código Fuente de una aplicación y los binarios, según unas reglas de seguridad definidas referentes al diseño y la codificación que son indicadores de la existencia de una vulnerabilidad. Las soluciones SAST analizan una aplicación desde adentro hacia afuera en un estado que no se ejecuta" (Gartner, 2019)

Los análisis SAST, por lo tanto, analizan el código de una aplicación desarrollada, siguiendo una serie de reglas que buscan patrones y flujos en el propio código fuente sin necesidad de compilar el código, que según unos estándares como CWE (Common Weakness Enumeration" determinan vulnerabilidades que se están produciendo al realizar una programación incorrecta. Este tipo de soluciones se pueden integrar fácilmente en los sistemas de integración continua, permitiendo monitorizar el código y detectar las vulnerabilidades que se van produciendo. Las pruebas de seguridad de las aplicaciones se crearon porque cuando construimos software y aplicaciones, la seguridad no siempre es lo primero en nuestras mentes.

SAST surgió por la falta de conciencia en seguridad, y el desconocimiento por parte de los desarrolladores de las implicaciones de un desarrollo inseguro, debido a los plazos rápidos, necesarios para el desarrollo y la innovación, pero dejando muchas vulnerabilidades de seguridad. Las herramientas SAST toman el control donde las personas no pueden llegar, y entregan resultados de manera inmediata para que se pueda actuar.

Las soluciones SAST, además soportan multitud de lenguajes y son capaces de detectar diferentes tipos de vulnerabilidades en distintos lenguajes, pero su inconveniente principal es la gran cantidad de falsos positivos que producen, que requieren de una fase de revisión y refinamiento de resultados.

Este tipo de soluciones además no suelen ser económicas, aunque existen algunas Open Source que dan buenos resultados: (gitlab, 2019)

Lenguaje	**Herramienta SAST**
.Net	Security Code Scan
C/C++	FlawFinder
Groovy	Find-sec-bugs
Java	Sonarqube
JavaScript	ESLint security plugin
PHP	Phpcs-security-audit
Python	Bandit
Múltiples lenguajes	Sonarqube

Tabla 1: Herramientas de seguridad SAST

3.3 ANÁLISIS DE DEPENDENCIAS

En general, cuando se desarrolla una aplicación se emplean librerías y dependencias que nos aportan funcionalidades esenciales para nuestro programa, desarrolladas por terceros y normalmente no disponemos de su código ni de su origen. En la gran mayoría de desarrollos una aplicación puede basar la mitad del desarrollo en librerías de terceros.

En el entorno de desarrollo de software de hoy en día, hay una gran cantidad de trabajo en colaboración con una gran comunidad de desarrolladores de código abierto y comunidades con muy poca comprensión de los problemas de seguridad que esto crea, por no hablar de las formas de gestionar este riesgo. Todos sabemos que no podemos dejar de usar código abierto, pero debemos tener consciencia de las implicaciones en cuanto a seguridad que puede desencadenar el uso de una librería vulnerable.

Por ello, aunque se realice un análisis estático de la seguridad del código completo podemos encontrarnos con vulnerabilidades que no tienen su origen en nuestro desarrollo, pero sí en una librería que hemos importado y que desconocíamos que era vulnerable.

Por ello es importante el uso de herramienta de análisis de dependencias, que por norma general analizan los gestores de dependencias empleados por nuestro software para la compilación, como puede ser un pom.xml en Maven, requirements.txt en Python, en busca de las librerías empeladas y posteriormente consultan en bases de datos de vulnerabilidades como el NIST, en busca de las vulnerabilidades ya conocidas para una versión concreta de una librería.

3.4 INTEGRACIÓN CONTINUA. ORQUESTADORES

En el marco que nos encontramos, la metodología DevOps, pero desde una perspectiva de seguridad DevSecOps, la integración continua es vital.

Para conseguir lo que se propone con DevSecOps, es necesario definir una serie de objetivos:

- Mejorar la frecuencia de desarrollo e integración de cambios.
- Índice de errores mínimos o nulos durante la integración.
- Tiempo de resolución de incidencias bajo.
- Asegurar la escalabilidad con cada iteración.
- Entregas rápidas y frecuentes al cliente.
- Coordinación entre equipos.

Uno de los puntos principales para alcanzar los objetivos que marca la metodología DevSecOps, es la coordinación del trabajo realizado por el equipo de desarrolladores. Es en este momento cuando entra en juego el concepto de integración continua, cuyo objetivo principal es coordinar e integrar el trabajo de todo el equipo de desarrollo de software en una línea principal de forma muy frecuente, y entregar el producto con estos nuevos cambios tan pronto sea posible.

Uno de los motores principales hoy en día para monitorización, entrega/integración continua es Jenkins.

3.4.1 JENKINS

Jenkins es una herramienta open Source de integración continua, usada principalmente para orquestar procesos en el desarrollo de software, pero sus grandes capacidades permiten utilizarlo para mas cosas.

- Orquestador de procesos
- Ejecuta tareas manuales o automáticas.
- Gran soporte para plugins, que añaden nuevas funcionalidades
- Fácil de usar
- Ejecución distribuida en agentes.

Jenkins actúa orquestando cada proceso y se lleva a cabo normalmente cada cierto tiempo cuya función principal es la descarga de las fuentes desde el control de versiones, su posterior compilación, la ejecución de pruebas y la generación de informes (entre otras muchas posibilidades).

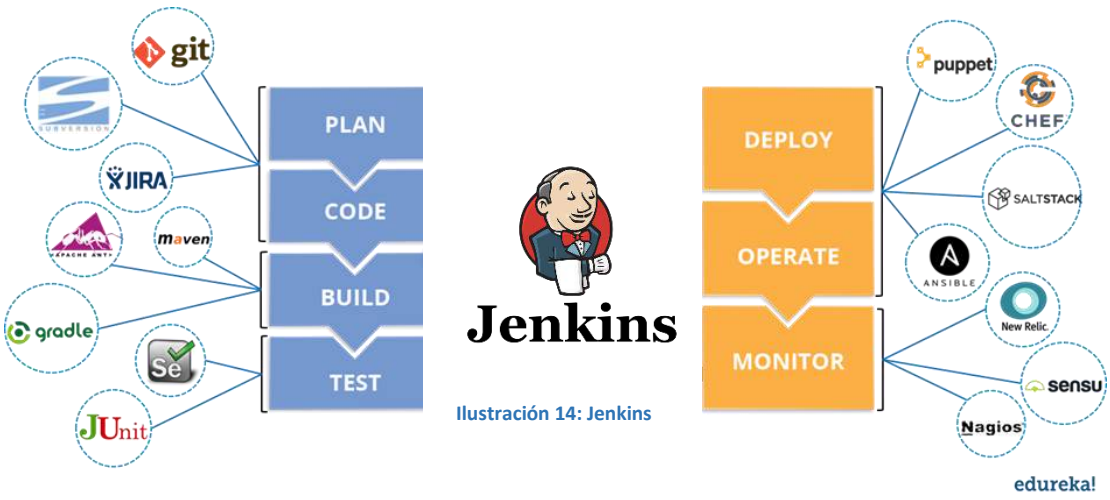

Ilustración 14: Jenkins

Como se observa en la anterior imagen, no solo se limita a la integración continua, sino que es capaz de abordar todas las fases del ciclo de vida del software, y unificar tanto el Desarrollo como las operaciones.

4. DESARROLLO DE LA SOLUCIÓN TÉCNICA

Uno de los objetivos de este trabajo de final de máster es el diseño de una solución que permita incluir la seguridad en todas las fases del ciclo de vida del software, para ello apoyándome en la cultura DevOps que como se ha mencionado con anterioridad nos permite mediante el cambio cultural de acercamiento entre equipos, y los pilares de automatización e integración continua, definir un ciclo de vida del desarrollo del software seguro DevSecOps.

4.1 DISEÑO DEL CICLO DE VIDA DEL SOFTWARE DEVSECOPS

En primer lugar, deben definirse las distintas fases de un ciclo de vida del software DevOps:

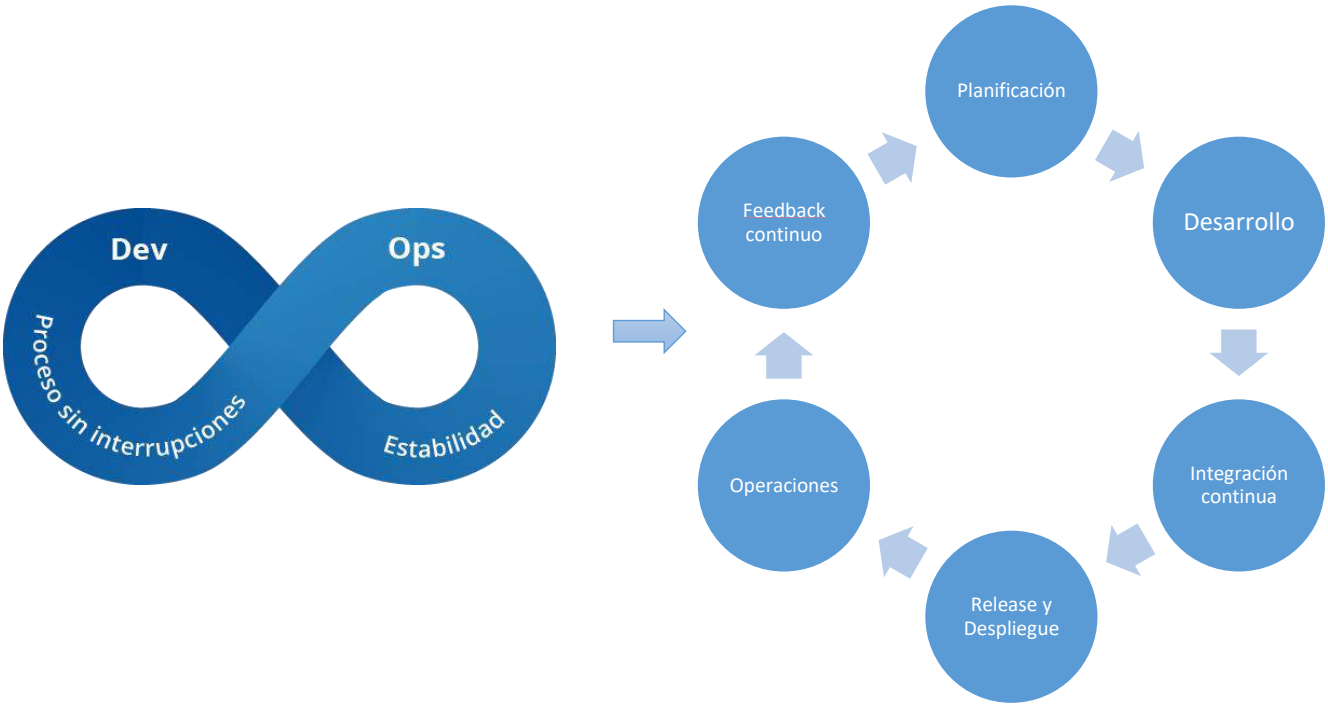

Diagrama 1: Conversión Ciclo DevOps

4.1.1 PLANIFICACIÓN

La primera fase del ciclo DevOps se basa en habilitar los mecanismos necesarios para dar la posibilidad a cualquier usuario de aportar de forma continua sus ideas y que estas puedan ser transformadas en requerimientos u objetivos, siendo estos priorizados e incluidos en próximas iteraciones en forma de historias de usuario.

En el modelo planteado en esta fase, la seguridad se incluye recogiendo en la fase de establecimiento de requisitos, también **requisitos de seguridad**, obteniendo información como la forma de autenticación, autorización, tipos de cifrado.

Por otro lado, se debe realizar un **modelado de amenazas** posibles frente a las que puede encontrarse y enfrentarse el desarrollo.

Por último, antes de comenzar la fase de codificación, se debe **formar a los** desarrolladores para que tengan conciencia de seguridad mientras se desarrolla.

Diagrama 2: Fase planificación. DevSecOps

4.1.2 DESARROLLO

Esta etapa en el proceso de desarrollo se realiza la implementación del código del software. Esta fase es crítica para la seguridad y para poder realizar una correcta monitorización del código, tener un manejo de los cambios y un control de repositorios debe subirse el código a un repositorio.

Previamente a la subida del código al repositorio, este modelo plantea **la codificación debe realizarse teniendo una conciencia de seguridad**, por parte de los desarrolladores fruto de la formación que recibieron previamente. Además, en **el IDE se propone realizar un análisis estático del código** en el momento del desarrollo, previa subida al repositorio. Esta fase no requiere un proceso de revisión de las vulnerabilidades detectadas en el código.

En el modelo planteado, el código se debe subir a un **repositorio** y en este momento se realizará un **análisis de seguridad de las dependencias** utilizadas.

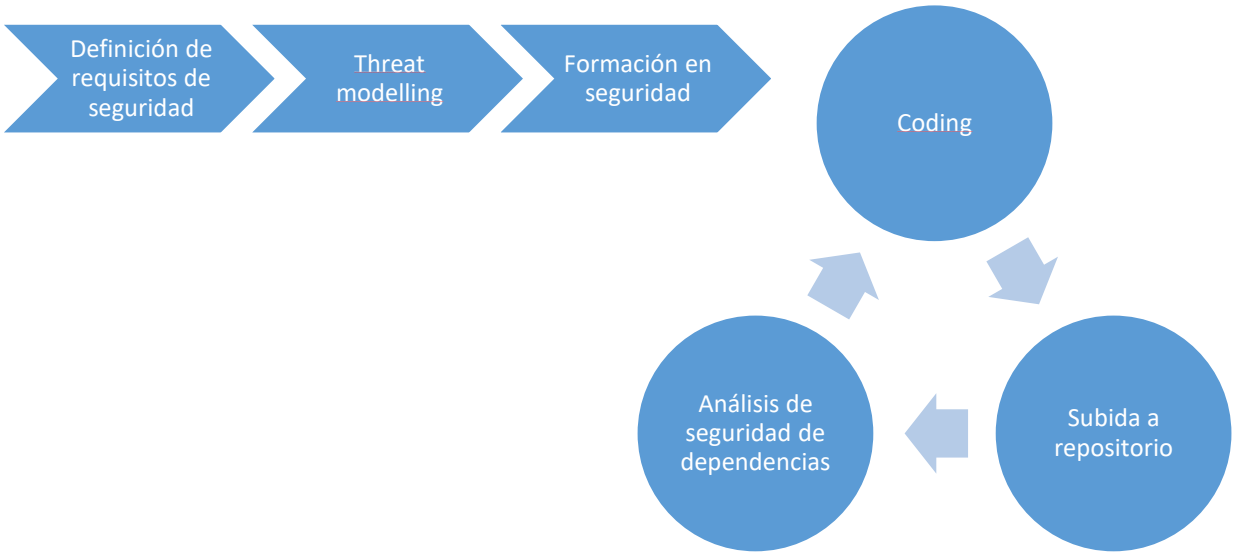

Diagrama 3: Fase Desarrollo. DevSecOps

4.1.3 INTEGRACIÓN CONTINUA

Como se mencionó en anteriores puntos de este trabajo, la integración continua es un proceso crítico en todos los ciclos DevOps. En esta fase del ciclo de vida se realiza el testing del código fuente subido al repositorio.

En el modelo planteado, se monitorizan los cambios subidos sobre el repositorio, en este momento se desencadena el análisis estático de seguridad en el código **SAST**.

Este análisis generará unos resultados de seguridad fruto del código que se encuentre en el repositorio, este código es descargado y subido a la aplicación SAST, esta analiza la seguridad y se obtienen los resultados del último desarrollo subido al repositorio.

De esta manera obtendremos en todo momento el **estado de la seguridad de la versión de la aplicación integrada en el repositorio.**

En esta fase, se pueden aplicar políticas de seguridad, dependiendo de los requisitos de seguridad definidos en la fase de planificación. En el caso de superar unos determinados umbrales de seguridad, el código no promocionará a la siguiente fase.

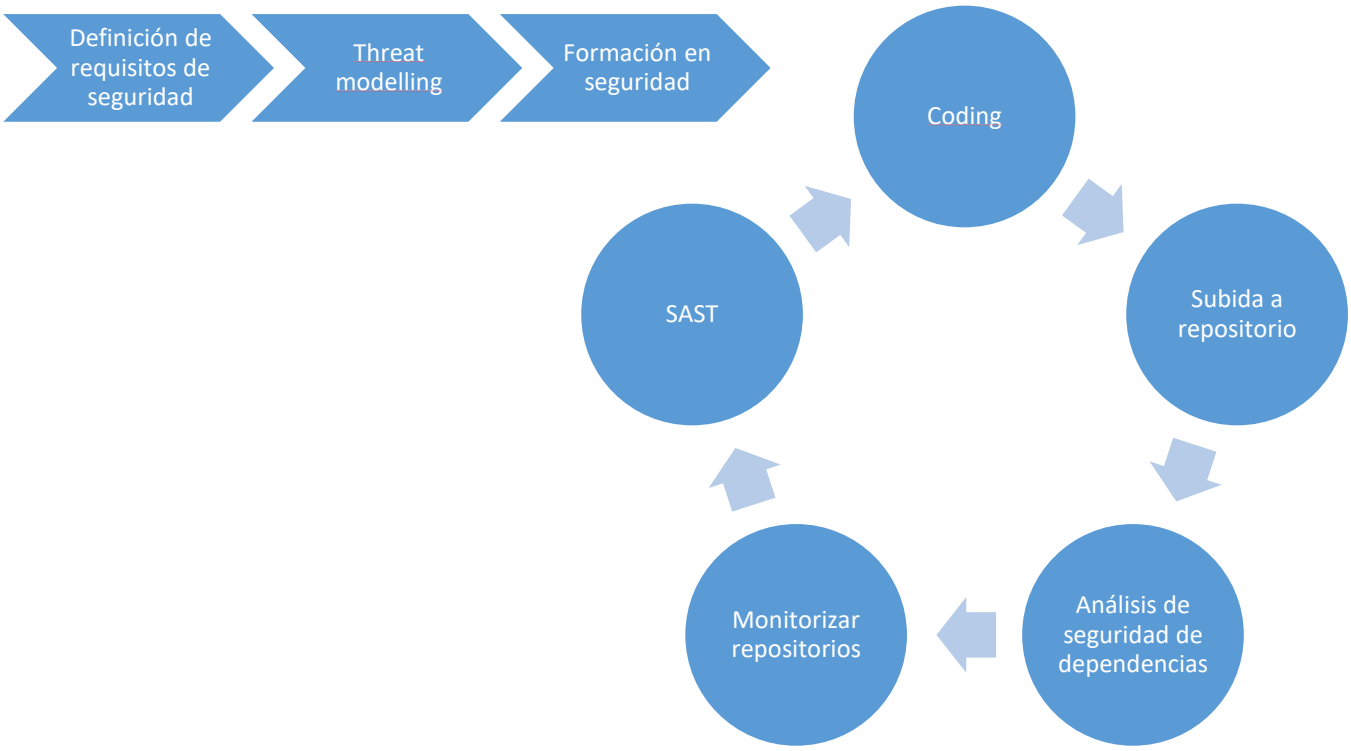

Diagrama 4: Fase de Integración continua. DevSecOps

4.1.4 RELEASE Y DESPLIEGUE

En esta fase del ciclo de vida del software, se realiza la construcción de la solución a partir del Código de una rama del repositorio y las dependencias necesarias para su correcto posterior despliegue en un entorno. Una vez construida la nueva versión de la aplicación, esta es desplegada en el entorno objetivo.

En el modelo planteado, la construcción de la nueva versión se realiza empleando **Docker**, de esta manera se genera **la nueva imagen Docker a partir del código** y las dependencias y la definición del **Dockerfile**.

Una vez se ha construido la nueva imagen que va a ser desplegada en un contenedor, se debe realizar un **análisis de la seguridad de la imagen Docker** empleando las herramientas destinadas para este tipo de análisis.

Si se cumplen los determinados niveles de seguridad definidos en la fase de planificación se realiza el **despliegue** del **contenedor Docker** con la imagen construida en la máquina objetivo,

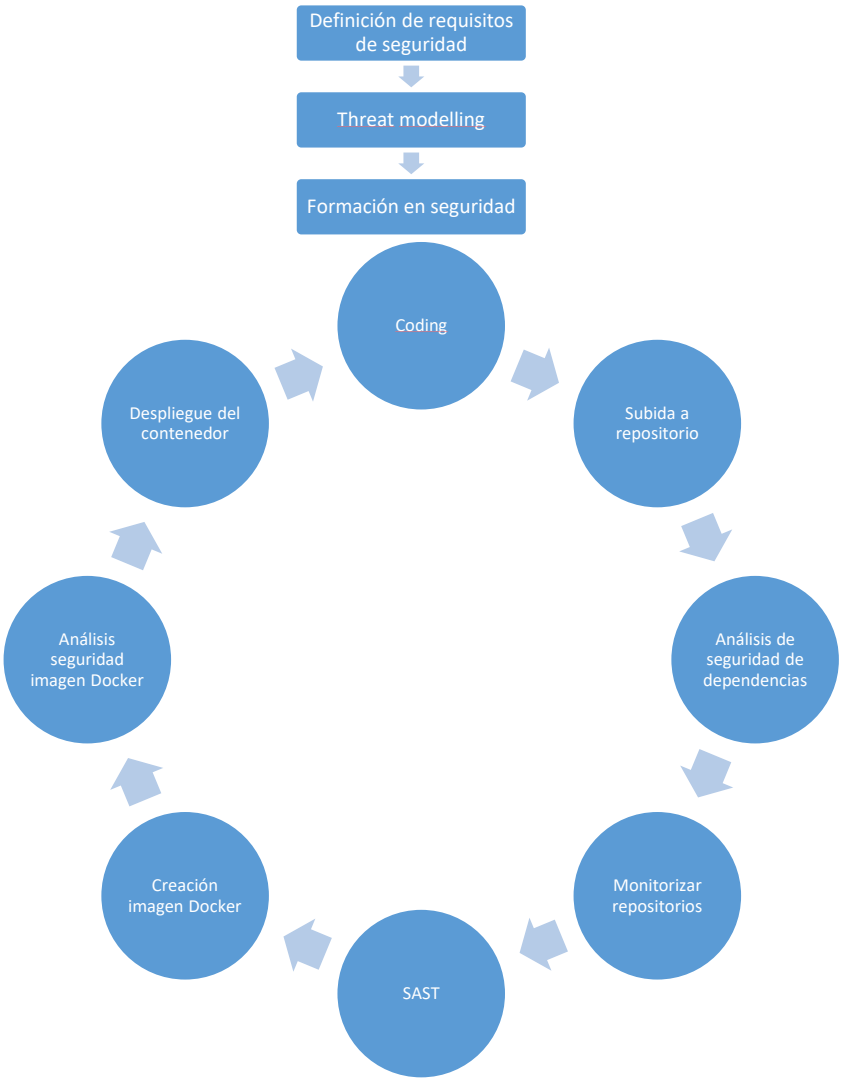

Diagrama 5: Fases de Release y despliegue. DevSecOps

4.1.5 OPERACIONES

En la fase de operaciones, la nueva versión o release de la aplicación se aplican los controles del sistema para asegurar que reúna los requisitos pedidos. Además, se realiza todas las fases de testing y QA para garantizar que el Sistema funciona correctamente según las especificaciones, en rendimiento, utilización y funcionalidad. Además, se realiza el mantenimiento del Sistema que aloja la nueva release.

En esta fase en el modelo propuesto, a parte de las pruebas de QA, se propone la realización de pruebas dinámicas de seguridad del Código **DAST**. Es decir, una vez la nueva versión de la aplicación se encuentra desplegada en el entorno de ejecuta un análisis DAST con un conjunto de pruebas automáticas previamente definidas para contemplar los requisitos de seguridad definidos.

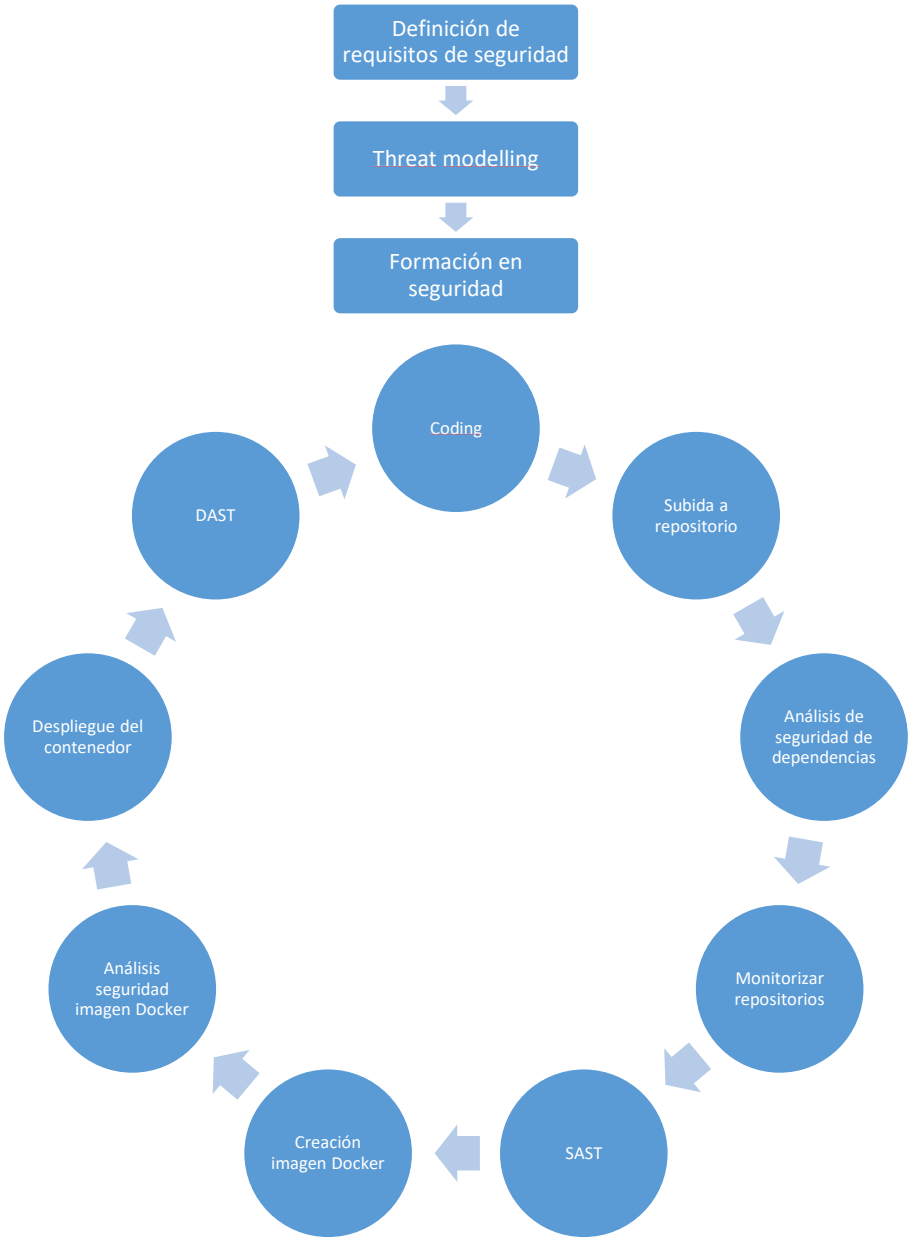

Diagrama 6: Fase de Operaciones. DevSecOps

4.1.6 FEEDBACK CONTINUO

Para lograr tener total transparencia entre las diferentes fases y equipos y conocer el estado del proyecto, en cada una de las fases se deben empelar sistemas de monitorización y reporte de resultados.

Por ello el empleo de herramientas como Splunk para monitorización de los logs de la aplicación desplegada, reporte de las vulnerabilidades a los equipos involucrados de los análisis SAST, DAST y de imágenes Docker. Por otro lado, se propone el empleo de un sistema de monitorización de los entornos utilizando Grafana.

Por último, todo este ciclo de vida es integrable dentro de un pipeline de Jenkins o cualquier otra herramienta de integración continua, permitiendo conocer el estado de cada una de las fases.

4.2 TECNOLOGÍAS EMPLEADAS

Las herramientas que quiero utilizar para el diseño de la solución técnica se basan en la premisa de ser Open Source:

- Definición de requisitos empleando metodología IEEE 830-1998

Ilustración 15: Estándar IEEE 830-1998

- Repositorio tipo Git. GitLab.

Ilustración 16: Gitlab

- Sistema de integración continua Jenkins.

Ilustración 17: Jenkins Logo

- Analizador de dependencias Owasp Dependency Check

Ilustración 18: Owasp Dependency Checker

- Analizador estático. Sonarqube

Ilustración 19:SonarQube

- Docker

Ilustración 20:Docker

- Análisis seguridad Docker clair

Ilustración 21:Clair

- DAST. Owasp ZAP.

Ilustración 22: Owasp ZAP

4.3 DIAGRAMA FINAL DEL DISEÑO CICLO DE VIDA DEVSECOPS

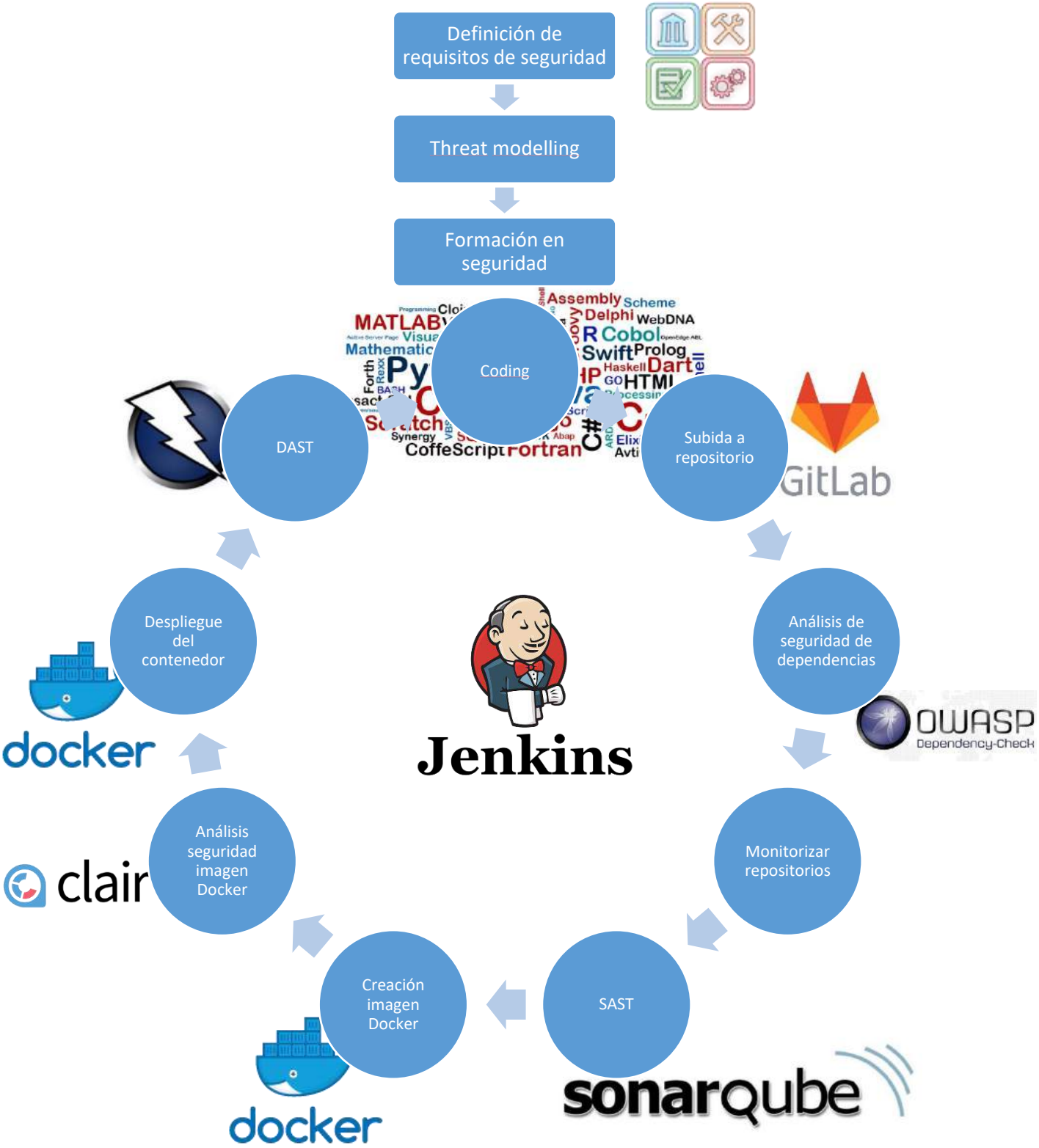

Diagrama 7: Diseño Final DevSecOps

4.4 DESARROLLO DE LA SOLUCIÓN

4.4.1 PLANIFICACIÓN. REQUISITOS DE SEGURIDAD DEL DESARROLLO SEGURO DEL SOFTWARE

Este documento presenta, en castellano, el formato de Especificación de Requisitos Software (ERS) según la última versión del estándar IEEE 830-1998. (gmendez, 2018)

4.4.1.1 FICHA DEL DOCUMENTO

Fecha	Revisión	Autor	Verificado dep. calidad.
01/04/2019	**V1.0.0**	**Sergio Iriz Ricote**	**Sergio Iriz Ricote**

Tabla 2: Ficha documento especificación de requisitos

4.4.1.2 PROPÓSITO

Este documento certifica los controles de seguridad que deben cumplir los desarrollos realizados en esta organización.

El Desarrollo del Código y posterior revisión de la seguridad del código debe cubrir las siguientes áreas:

- Autenticación
- Autorización
- Gestión de Cookies
- Validación de Entrada de Datos
- Gestión de Errores / Fuga de Información
- Log / Auditoría
- Cifrado de Datos
- Gestión de Sesiones (Login/Logout)

Este documento va dirigido a todos los equipos involucrados en el ciclo de vida del desarrollo del software de esta organización. Desde los equipos de desarrollo, operaciones y seguridad.

4.4.1.3 REQUISITOS ESPECÍFICOS

4.4.1.3.1 REQUISITOS DE SEGURIDAD

4.4.1.3.1.1 REQUERIMIENTOS DE SEGURIDAD DE LA INFORMACIÓN

Número de requisito	RS1		
Nombre de requisito	Gestión de usuarios y roles		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	El sistema debe permitir la Gestión de Seguridad de Usuarios, grupos de usuarios y asignación de Roles y perfiles de usuarios, permitiendo asociar las acciones disponibles en el sistema a roles de usuario, permitiendo parametrizar las funcionalidades que cada actor puede usar en el sistema, los permisos de acceso al sistema para los usuarios podrán ser cambiados solamente por el administrador de acceso a datos.		
Prioridad del requisito	☒ Alta/Esencial	☐ Media/Deseado	☐ Baja/ Opcional

Tabla 3: Requisito RS1

Número de requisito	RS2		
Nombre de requisito	Registro de sesiones de usuario		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	La aplicación al ingresar al usuario debe mostrar la última fecha y hora de ingreso y debe garantizar solo una sesión por usuario.		
Prioridad del requisito	☒ Alta/Esencial	☐ Media/Deseado	☐ Baja/ Opcional

Tabla 4: Requisito RS2

Número de requisito	RS3		
Nombre de requisito	Control de permisos de sesión del usuario		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	Un usuario puede estar asociado a uno o más roles, de tal manera que los menús de navegación del sistema se muestran o despliegan dependiendo de las acciones asociadas a cada rol de usuario, permitiendo así que cuando el usuario es autenticado correctamente el sistema verifica los roles que tiene activos para otorgarle únicamente las acciones autorizadas a realizar.		
Prioridad del requisito	☒ Alta/Esencial	☐ Media/Deseado	☐ Baja/ Opcional

Tabla 5: Requisito RS3

Número de requisito	RS4		
Nombre de requisito	Registro de usuarios del sistema		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	Las aplicaciones deben generar informe con los usuarios activos, los perfiles de los mismos, los usuarios inactivos con sus fechas de bajas y altas de la aplicación a fin de hacer auditorías de usuarios periódicamente.		
Prioridad del requisito	☐ Alta/Esencial	☒ Media/Deseado	☐ Baja/ Opcional

Tabla 6: Requisito RS4

Número de requisito	RS5		
Nombre de requisito	Integración con LDAP		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	El sistema debe integrarse con LDAP para los procesos de inicio de sesión y autenticación.		
Prioridad del requisito	☐ Alta/Esencial	☒ Media/Deseado	☐ Baja/ Opcional

Tabla 7: Requisito RS5

Número de requisito	RS6		
Nombre de requisito	Confidencialidad		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	El sistema debe asegurar los aspectos de la transacción, asegurando la información de autenticación secreta de usuario, validando y verificando que la transacción permanezca confidencial y que se mantenga la privacidad asociada con todas las partes involucradas.		
Prioridad del requisito	☒ Alta/Esencial	☐ Media/Deseado	☐ Baja/ Opcional

Tabla 8: Requisito RS6

Número de requisito	RS7		
Nombre de requisito	Bloqueo de sesión		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	El sistema debe incluir controles de bloqueo de cuenta después de un máximo de 5 intentos erróneos a fin de evitar ataques de fuerza bruta.		
Prioridad del requisito	☐ Alta/Esencial	☐ Media/Deseado	☒ Baja/ Opcional

Tabla 9: Requisito RS7

Número de requisito	RS8		
Nombre de requisito	Cumplimiento de la ley de protección de datos		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	El sistema debe garantizar el cumplimiento de la normatividad vigente en cuanto a protección de datos personales, debe permitir el manejo de excepciones previa autorización de los usuarios finales, cuando los sistemas de información soliciten datos personales al usuario final se debe establecer un mecanismo que permita registrar que se ha autorizado o no el tratamiento de los mismos.		
Prioridad del requisito	☒ Alta/Esencial	☐ Media/Deseado	☐ Baja/ Opcional

Tabla 10: Requisito RS7

4.4.1.3.1.2 REQUERIMIENTOS DE SEGURIDAD EN LAS COMUNICACIONES

Número de requisito	RS9		
Nombre de requisito	Cifrado de las comunicaciones		
Tipo	☐ Requisito	☒ Restricción	
Descripción del requisito	Las comunicaciones con el servidor deben realizarse a través del protocolo seguro de comunicaciones HTTPS.		
Prioridad del requisito	☒ Alta/Esencial	☐ Media/Deseado	☐ Baja/ Opcional

Tabla 11: Requisito RS9

Número de requisito	RS10		
Nombre de requisito	Protección Man in the Middle		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	Se debe verificar el certificado SSL empleado en las comunicaciones, con técnicas como SSL Pinning		
Prioridad del requisito	☐ Alta/Esencial	☒ Media/Deseado	☐ Baja/ Opcional

Tabla 12: Requisito RS10

Número de requisito	RS11		
Nombre de requisito	Uso de tokens de sesión únicos		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	Se deben emplear token de sesión únicos en las comunicaciones, para evitar la suplantación ni duplicado de peticiones.		
Prioridad del requisito	☒ Alta/Esencial	☐ Media/Deseado	☐ Baja/ Opcional

Tabla 13: Requisito RS11

4.4.1.3.1.3 REQUERIMIENTOS DE DESARROLLO SEGURO

Número de requisito	RS12		
Nombre de requisito	Uso de sistemas de control de versiones		
Tipo	☐ Requisito	☒ Restricción	
Descripción del requisito	Todos los desarrollos de la organización deben desarrollarse empelando repositorios como Git, SVN o Dimensions.		
Prioridad del requisito	☒ Alta/Esencial	☐ Media/Deseado	☐ Baja/ Opcional

Tabla 14: Requisito RS12

Número de requisito	RS13		
Nombre de requisito	Envío de credenciales de autenticación mediante POST		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	No se podrán realizar peticiones de autenticación empleando métodos GET.		
Prioridad del requisito	☐ Alta/Esencial	☒ Media/Deseado	☐ Baja/ Opcional

Tabla 15: Requisito RS13

Número de requisito	RS14		
Nombre de requisito	Validación de datos de entrada		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	Debe realizarse una validación de todos los datos que pueda manipular un usuario		
Prioridad del requisito	☒ Alta/Esencial	☐ Media/Deseado	☐ Baja/ Opcional

Tabla 16: Requisito RS14

Número de requisito	RS15		
Nombre de requisito	Credenciales cifradas		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	Asegurar que las credenciales de autenticación no van en claro, ni se encuentren hardcodeadas en el código.		
Prioridad del requisito	☒ Alta/Esencial	☐ Media/Deseado	☐ Baja/ Opcional

Tabla 17: Requisito RS15

Número de requisito	RS16		
Nombre de requisito	Eliminación de hardcodeos en el código		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	Todos los valores deben estar parametrizados en ficheros de configuración y nunca hardcodeadas en el código		
Prioridad del requisito	☐ Alta/Esencial	☒ Media/Deseado	☐ Baja/ Opcional

Tabla 18: Requisito RS16

Número de requisito	RS17		
Nombre de requisito	Uso de algoritmos de cifrados seguros		
Tipo	☐ Requisito	☒ Restricción	
Descripción del requisito	Uso de algoritmos de cifrados seguros como AES, RSA, SHA-2, SHA-3. No usar MD5, SHA-1, TDEA, MD4, algoritmo con clave menor de 112 bits.		
Prioridad del requisito	☒ Alta/Esencial	☐ Media/Deseado	☐ Baja/ Opcional

Tabla 19: Requisito RS17

Número de requisito	RS18		
Nombre de requisito	Validación cabeceras http		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	Asegurarnos de validar todos los campos, cookies, http headers/bodies y form fields.		
Prioridad del requisito	☒ Alta/Esencial	☐ Media/Deseado	☐ Baja/ Opcional

Tabla 20: Requisito RS18

Número de requisito	RS19		
Nombre de requisito	Validación datos de entrada y salida en base de datos		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	A nivel de la base de datos debe poder definirse reglas de validación de los datos, así como el uso de procedimientos almacenados o prepare Statements para garantizar la seguridad e integridad de la información de la base de datos.		
Prioridad del requisito	☒ Alta/Esencial	☐ Media/Deseado	☐ Baja/ Opcional

Tabla 21: Requisito RS19

Número de requisito	RS20		
Nombre de requisito	Validación en el servidor		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	Realización de la validación de todos los datos en el lado servidor.		
Prioridad del requisito	☒ Alta/Esencial	☐ Media/Deseado	☐ Baja/ Opcional

Tabla 22: Requisito RS20

4.4.1.3.1.4 REQUERIMIENTOS DE GESTIÓN DE ERRORES

Número de requisito	RS21		
Nombre de requisito	Control de errores		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	Asegurar que todas las llamadas a métodos/funciones que devuelven un valor tienen su control de errores y además se comprueba el valor devuelto.		
Prioridad del requisito	☒ Alta/Esencial	☐ Media/Deseado	☐ Baja/ Opcional

Tabla 23: Requisito RS21

Número de requisito	RS22		
Nombre de requisito	Uso de log		
Tipo	☐ Requisito	☒ Restricción	
Descripción del requisito	Evitar el uso de funciones que muestren la pila de llamadas o de memoria como printstacktrace. Uso de errores controlados mediante logs.		
Prioridad del requisito	☐ Alta/Esencial	☒ Media/Deseado	☐ Baja/ Opcional

Tabla 24: Requisito RS22

Número de requisito	RS23		
Nombre de requisito	Mostrar errores controlados al usuario		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	Mostrar errores controlados al usuario para evitar disclosures de información del sistema empelado.		
Prioridad del requisito	☐ Alta/Esencial	☒ Media/Deseado	☐ Baja/ Opcional

Tabla 25: Requisito RS23

Número de requisito	RS24		
Nombre de requisito	Análisis de dependencias de terceros		
Tipo	☒ Requisito	☐ Restricción	
Descripción del requisito	No utilizar librerías de terceros con vulnerabilidades		
Prioridad del requisito	☐ Alta/Esencial	☒ Media/Deseado	☐ Baja/ Opcional

Tabla 26: Requisito RS24

4.4.2 DESARROLLO DEL CICLO DE VIDA DEL DESARROLLO DEL SOFTWARE DEVSECOPS

En esta sección se detallan los pasos seguidos para la implementación del Ciclo de vida del desarrollo del software DevSecOps objetivo.

4.4.2.1 INSTALACIÓN DE LAS HERRAMIENTAS

4.4.2.1.1 JENKINS

El sistema de integración continua elegido como comenté anteriormente es Jenkins, ya que se trata de una herramienta open Source con gran soporte por parte de la comunicad y que nos permite el desarrollo de pipelines para la integración de todas las fases del SDLC.

Para su instalación, en primera instancia, se ha descargado el instalador desde los repositorios oficiales. Como el sistema operativo donde se ha instalado se trata de Linux Ubuntu, en primer lugar, añadimos los repositorios a los repositorios del sistema operativo.

Ilustración 23: Descarga repositorios Jenkins

Una vez disponemos de los repositorios, podemos instalar Jenkins:

```
root@siriz-VirtualBox:/home/siriz/Escritorio/jenkins# sudo apt install jenkins
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
Se instalarán los siguientes paquetes adicionales:
  daemon
Se instalarán los siguientes paquetes NUEVOS:
  daemon jenkins
0 actualizados, 2 nuevos se instalarán, 0 para eliminar y 220 no actualizados.
Se necesita descargar 76,8 MB de archivos.
Se utilizarán 77,7 MB de espacio de disco adicional después de esta operación.
¿Desea continuar? [S/n]
```

Ilustración 24: Comando instalación Jenkins

Tras su finalización, se comprueba que el servicio está ejecutándose correctamente sobre el puerto 80.

```
root@siriz-VirtualBox:/home/siriz/Escritorio/jenkins# systemctl status jenkins
● jenkins.service - LSB: Start Jenkins at boot time
   Loaded: loaded (/etc/init.d/jenkins; bad; vendor preset: enabled)
   Active: active (exited) since mié 2019-04-17 00:21:51 CEST; 20s ago
     Docs: man:systemd-sysv-generator(8)

abr 17 00:21:46 siriz-VirtualBox systemd[1]: Starting LSB: Start Jenkins at boot time...
abr 17 00:21:47 siriz-VirtualBox jenkins[4960]: Correct java version found
abr 17 00:21:47 siriz-VirtualBox jenkins[4960]:  * Starting Jenkins Automation Server jenkins
abr 17 00:21:47 siriz-VirtualBox su[5004]: Successful su for jenkins by root
abr 17 00:21:47 siriz-VirtualBox su[5004]: + ??? root:jenkins
abr 17 00:21:47 siriz-VirtualBox su[5004]: pam_unix(su:session): session opened for user jenkins by (uid=0)
abr 17 00:21:51 siriz-VirtualBox jenkins[4960]:    ...done.
abr 17 00:21:51 siriz-VirtualBox systemd[1]: Started LSB: Start Jenkins at boot time.
root@siriz-VirtualBox:/home/siriz/Escritorio/jenkins#
```

Ilustración 25: Jenkins comprobación funcionamiento

A continuación, se seleccionan los plugins base que queremos instalar en nuestro Jenkins:

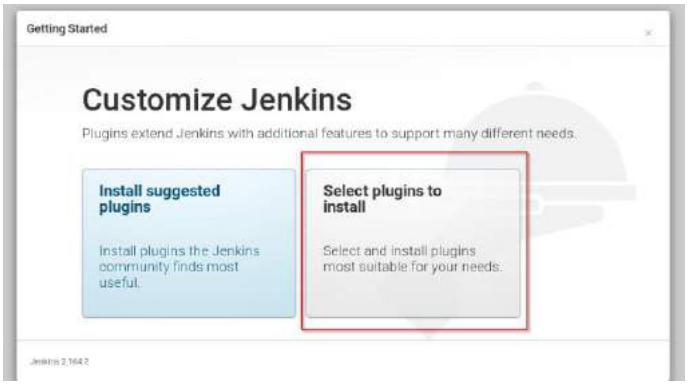

Ilustración 26: Jenkins funcionando

Algunos de los más importantes elegidos, son los plugins que den soporte a los distintos tipos de repositorio como Gitlab y Svn. Además, se debe instalar el plugin para la creación de pipelines, así como los steps condicionales por si fueran de utilidad:

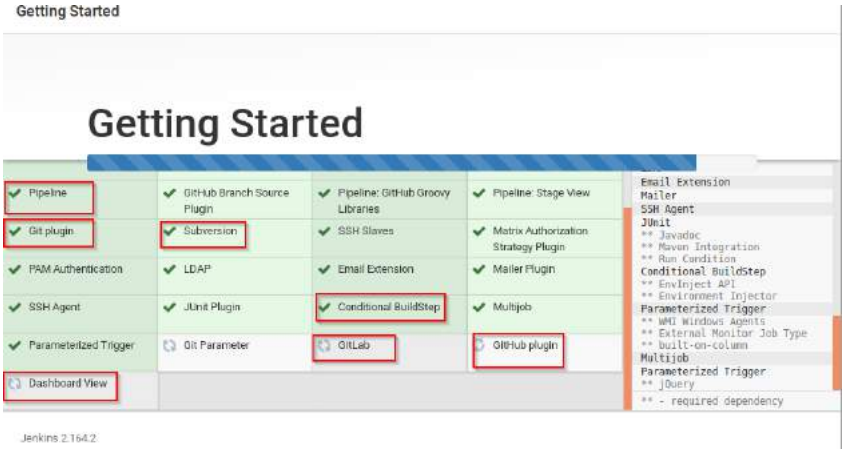

Ilustración 27: Plugins Jenkins

4.4.2.1.2 SONARQUBE

Para instalar Sonar, se ha decidido utilizar Docker debido a su simplicidad y las posibilidades que da de cara a escalarlo a distintas máquinas y posibles agentes de Jenkins.

Para ello se ha utilizado el siguiente fichero docker-compose.yml, que instala dentro de la misma subred Sonarqube escuchando en el puerto 9000 y su base de datos de tipo postgres en el puerto 5432, sobre la que únicamente se tiene visibilidad si nos encontramos en la misma subred. Como se observa también se han usado volúmenes para lograr una persistencia de los datos y una mayor accesibilidad:

```
version: "2"

services:
  sonarqube:
    image: sonarqube
    expose:
      - 9000
    ports:
      - "127.0.0.1:9000:9000"
    networks:
      - sonarnet
    environment:
      - SONARQUBE_JDBC_URL=jdbc:postgresql://db:5432/sonar
      - SONARQUBE_JDBC_USERNAME=sonar
      - SONARQUBE_JDBC_PASSWORD=sonar
    volumes:
      - sonarqube_conf:/opt/sonarqube/conf
      - sonarqube_data:/opt/sonarqube/data
      - sonarqube_extensions:/opt/sonarqube/extensions
      - sonarqube_bundled-plugins:/opt/sonarqube/lib/bundled-plugins

  db:
    image: postgres
    networks:
      - sonarnet
    environment:
      - POSTGRES_USER=sonar
      - POSTGRES_PASSWORD=sonar
    volumes:
      - postgresql:/var/lib/postgresql
      - postgresql_data:/var/lib/postgresql/data

networks:
  sonarnet:
    driver: bridge

volumes:
  sonarqube_conf:
```

```
volumes:
  sonarqube_conf:
  sonarqube_data:
  sonarqube_extensions:
  sonarqube_bundled-plugins:
  postgresql:
  postgresql_data:
```

Ilustración 28: docker-compose SonarQube-Postgres

Ejecutando el comando docker-compose up, se comenzarán a desplegar tanto sonar como su base de datos:

```
Attaching to sonarqube_sonarqube_1, sonarqube_db_1
db_1          | The files belonging to this database system will be owned by user "postgres".
db_1          | This user must also own the server process.
db_1          |
db_1          | The database cluster will be initialized with locale "en_US.utf8".
db_1          | The default database encoding has accordingly been set to "UTF8".
db_1          | The default text search configuration will be set to "english".
db_1          |
db_1          | Data page checksums are disabled.
db_1          |
db_1          | fixing permissions on existing directory /var/lib/postgresql/data ... ok
db_1          | creating subdirectories ... ok
db_1          | selecting default max_connections ... 100
db_1          | selecting default shared_buffers ... 128MB
db_1          | selecting dynamic shared memory implementation ... posix
db_1          | creating configuration files ... ok
db_1          | running bootstrap script ... ok
db_1          | performing post-bootstrap initialization ... ok
sonarqube_1   | 17:14:50.747 [main] WARN org.sonar.application.config.AppSettingsLoaderImpl - Configuration file not found: /o
be/conf/sonar.properties
db_1          | syncing data to disk ... ok
db_1          |
db_1          | WARNING: enabling "trust" authentication for local connections
db_1          | You can change this by editing pg_hba.conf or using the option -A, or
db_1          | --auth-local and --auth-host, the next time you run initdb.
db_1          |
db_1          | Success. You can now start the database server using:
db_1          |
db_1          |     pg_ctl -D /var/lib/postgresql/data -l logfile start
db_1          |
db_1          | waiting for server to start....2019-04-16 17:14:51.705 UTC [40] LOG:  listening on Unix socket "/var/run/postg
GSQL.5432"
db_1          | 2019-04-16 17:14:51.807 UTC [41] LOG:  database system was shut down at 2019-04-16 17:14:47 UTC
db_1          | 2019-04-16 17:14:51.848 UTC [40] LOG:  database system is ready to accept connections
db_1          |  done
db_1          | server started
sonarqube_1   | 2019.04.16 17:14:53 INFO  app[][o.s.a.AppFileSystem] Cleaning or creating temp directory /opt/sonarqube/temp
```

Ilustración 29: Ejecución docker-compose Sonar

Una vez se ha finalizado la instalación, comprobamos que efectivamente, disponemos de SonarQube a través del puerto 9000

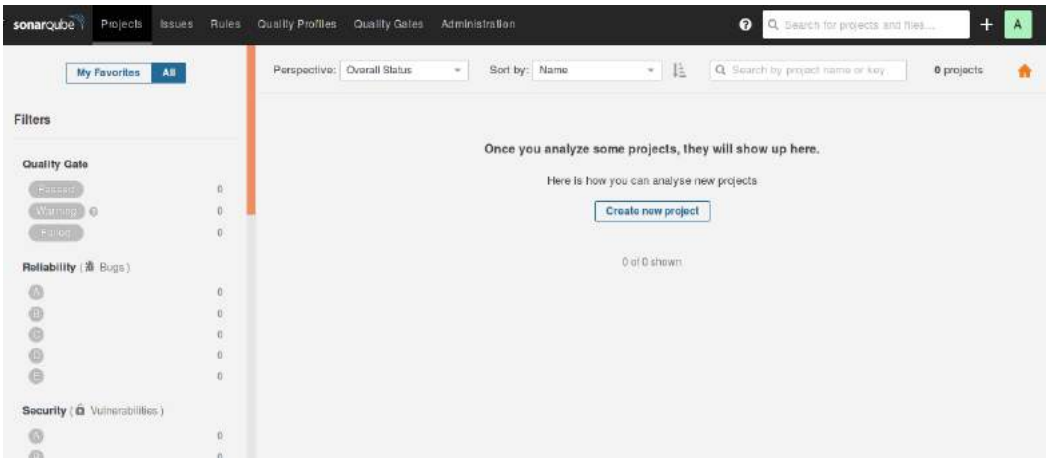

Ilustración 30: Sonarqube instalado

4.4.2.1.3 ZAP PROXY

En primer lugar, se ha descargado el código de instalación desde la página oficial de Owasp ZAP.

Una vez descargado, descomprimido y movido a opt:

```
root@siriz-VirtualBox:/home/siriz/Escritorio# mv ZAP_2.7.0 /opt/
root@siriz-VirtualBox:/home/siriz/Escritorio# cd /opt/
root@siriz-VirtualBox:/opt# ls
APP      CAS      containerd  got_certinfo.ldif  perfStats.log  sublime_text              webserver.cer
cacerts  cas.log  entrega     java-jdk           restart.sh     VBoxGuestAdditions-5.1.30  ZAP_2.7.0
root@siriz-VirtualBox:/opt# ls -lrt
total 420
drwxr-xr-x 4 root  root    4096 oct 22  2017 sublime_text
drwxr-xr-x 9 root  root    4096 ene 23  2018 VBoxGuestAdditions-5.1.30
drwxr-x--- 9 root  root    4096 may  8  2018 APP
-rwxr-x--- 1 root  root  197954 may  8  2018 cacerts
drwxr-x--- 9 root  root    4096 may  8  2018 CAS
-rwxr-x--- 1 root  root     289 may  8  2018 got_certinfo.ldif
drwxr-x--- 3 root  root    4096 may  8  2018 java-jdk
-rwxr-x--- 1 root  root     110 may  8  2018 restart.sh
-rwxr-x--- 1 root  root     901 may  8  2018 webserver.cer
drwxr-x--- 2 root  root    4096 may  8  2018 entrega
-rw-r----- 1 root  root   25179 may  8  2018 cas.log
-rwxr-x--- 1 root  root  153582 may  8  2018 perfStats.log
drwx--x--x 4 root  root    4096 abr 15 19:16 containerd
drwxrwxr-x 9 siriz siriz   4096 abr 17 00:36 ZAP_2.7.0
root@siriz-VirtualBox:/opt# chown -R root:root ZAP_2.7.0/
```

Ilustración 31: Instalación ZAP Proxy

Se instala el plugin que da soporte a ZAP desde los pipelines de Jenkins. Esto nos permitirá ejecutar mediante las llamadas de la librería del pipeline ZAP desde un propio paso del pipeline:

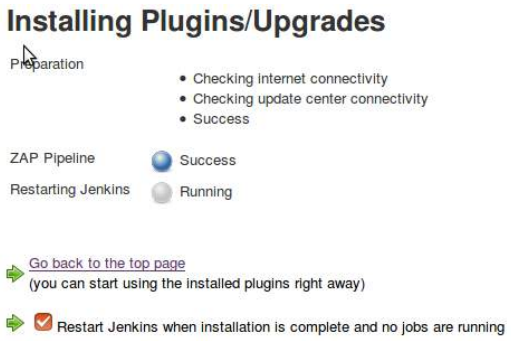

Ilustración 32: Instalación plugin de Jenkins, ZAP

Para comprobar su funcionamiento, se ha implementado un pequeño pipeline, que inicializa ZAP en el puerto 9091:

```
pipeline {
    agent any
    stages {
        stage('Setup') {
            steps {
                script {
                    startZap(host: "127.0.0.1", port: 9091, timeout:500, zapHome:
"/opt/ZAP_2.7.0")
                }
            }
        }
    }
}
```

Como se muestra en la siguiente imagen, ZAP se despliega en el puerto 9091, ejecutando el script de despliegue descargado previamente con el código:

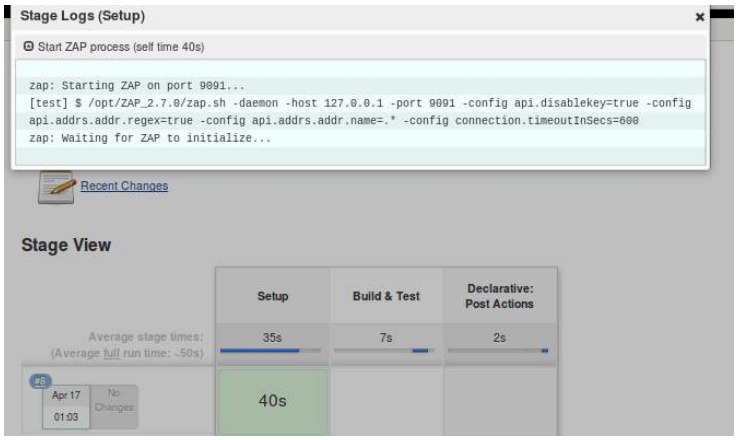

Ilustración 33: Prueba ejecución pipeline ZAP

4.4.2.1.4 GITLAB

Un elemento esencial para poder desarrollar un sistema de integración continua y detectar los cambios que vayamos realizando sobre el código para iniciar un ciclo de vida del software, es la instalación de un repositorio que permita una correcta integración de manera ágil con un pipeline de Jenkins. Por ello se ha escogido Gitlab.

Gitlab ha sido instalado empleando Docker a partir de la imagen oficial y el siguiente comando Docker run:

```
sudo docker run -d \
  --hostname gitlab.tfm.com \
  --publish 443:443 --publish 80:80 --publish 22:22 \
  --name gitlab \
  --restart always \
  gitlab/gitlab-ce:latest
```

```
CONTAINER ID        IMAGE                    COMMAND                  CREATED             STATUS
                                                                      NAMES
5caf84a9be85        gitlab/gitlab-ce:latest  "/assets/wrapper"        21 seconds ago      Up 13 seconds
2->22/tcp, 0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp   gitlab
1105b103cb49        postgres                 "docker-entrypoint.s…"   4 hours ago         Up 4 hours
                                                                      sonarqube_db_1
3b126c4f0615        sonarqube                "./bin/run.sh"           4 hours ago         Up 4 hours
:9000->9000/tcp                                                       sonarqube_sonarqube_1
```

Ilustración 34: Contenedores ejecutando

Como se puede observar en este momento, disponemos de tres contenedores ejecutándose, Gitlab, Sonarqube y postgres.

4.4.2.1.5 CLAIR

Se dispone de un repositorio de Docker, un Docker Registry en el puerto 4443 por https empleando un certificado auto firmado.

Este repositorio será empleado para subir las imágenes Docker una vez se ha analizado la seguridad de una imagen Docker creada en el proceso de construcción.

Configuración, puerto expuesto 6060 para el api de clair.

Para la instalación de Clair, se ha creado una carpeta para guardar la configuración de Clair:

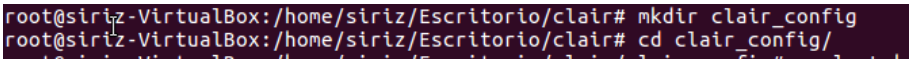

Ilustración 35: Instalación Clair

En esta carpeta, se ha creado el siguiente fichero de configuración, donde se indica esencialmente, donde se aloja la base de datos que utiliza Clair, así como el puerto sobre el que se expone el api de Clair con la que interactuará el pipeline de Jenkins.

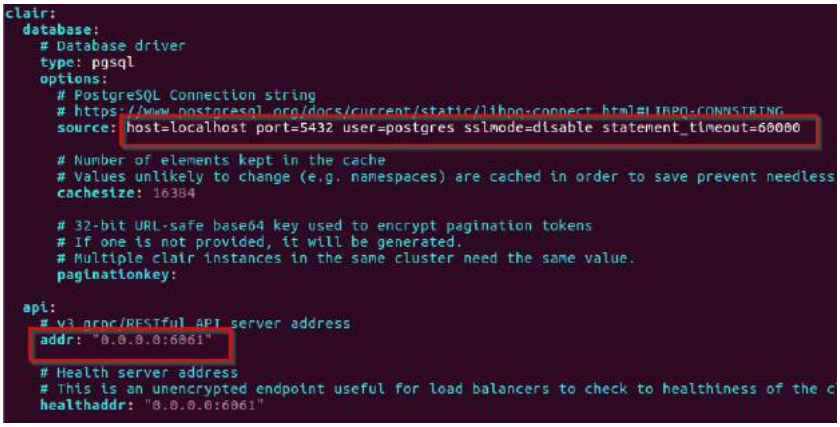

Ilustración 36: Configuración Clair

Una vez creado el fichero de configuración, se despliega en primer lugar la base de datos, en este caso postgres sobre el puerto 5432:

```
docker run --name clair_postgres -d -e POSTGRES_PASSWORD="" -p 5432:5432 postgres:9.6
```

A continuación, se deplegará mediante docker nuevamente Clair, mapeando en un volumen el fichero de configuración y exponiedno el puerto 6061 de su api:

```
docker run --name clair  -d --net=host -p 6060:6060 -p 6061:6061 -v /home/siriz/Escritorio/clair/clair_config:/config
quay.io/coreos/clair:v2.0.1 -config=/config/config.yaml
```

En este momento, disponemos de 6 contenedores ejecutándose con dos bases de datos Postgres, Clair, Gitlab, Sonarqube y el repositorio de Docker, Docker Registry:

```
CONTAINER ID        IMAGE                          COMMAND                  CREATED             STATUS                  PORTS                                                                                  NAMES
6856f819de83        quay.io/coreos/clair:v2.0.1    "/clair -config=/con…"   12 days ago         Up 4 days                                                                                                      clair
ce9fee2a1774        postgres:9.6                   "docker-entrypoint.s…"   12 days ago         Up 4 days               0.0.0.0:5432->5432/tcp                                                                 clair_postgres
48ef3053bd17        registry:2                     "/entrypoint.sh /etc…"   12 days ago         Up 4 days               5000/tcp, 0.0.0.0:443->443/tcp                                                         registry
5caf04a9be85        gitlab/gitlab-ce:latest        "/assets/wrapper"        13 days ago         Up 4 days (healthy)     0.0.0.0:22->22/tcp, 0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp                           gitlab
1105b103cb49        postgres                       "docker-entrypoint.s…"   2 weeks ago         Up 4 days               5432/tcp                                                                               sonarqube_db_1
3b126c4f0615        sonarqube                      "./bin/run.sh"           2 weeks ago         Up 4 days               127.0.0.1:9000->9000/tcp                                                               sonarqube_sonarqube_1
```

Ilustración 37: Contendores ejecutando 2

4.4.2.1.6 KLAR

Para facilitar la integración con el pipeline de Jenkins se ha utilizado una herramienta OpenSource, que ejecuta peticiones directamente sobre el api de Clair, y se ejecuta desde el terminal con diversas opciones de configuración. Esta herramienta es Klar.

Klar es una herramienta desarrollada en GoLang, por lo que, en primer lugar, debí instalarlo en el sistema operativo y posteriormente descargar Klar:

```
root@siriz-VirtualBox:/home/siriz/Escritorio/klar# go get github.com/optiopay/klar
```

Ilustración 38: Instalación Klar

Para comprobar el funcionamiento de Klar y la interacción con Clair se ejecutó la siguiente prueba, en las opciones de configuración, se especifica el puerto de Clair, el threshold que indica el número de vulnerabilidades a partir del cual se considera una imagen insegura, como el output de criticidades que se quiere mostrar. Para la prueba se analizó la imagen de postgres:

```
root@siriz-VirtualBox:/home/siriz/Escritorio/klar# CLAIR_ADDR=localhost:6060 CLAIR_OUTPUT=High CLAIR_THRESHOLD=10 klar postgres
clair timeout 1m0s
docker timeout: 1m0s
no whitelist file
Analysing 14 layers
Got results from Clair API v1
Found 0 vulnerabilities
```

Ilustración 39: Ejecución Klar

4.4.2.1.7 OWASP DEPENDENCY CHECK

Por último, Jenkins tiene integración con Owasp Dependency Check a partir de un plugin. Por lo que simplemente, se ha instalado en Jenkins.

Ilustración 40: Plugin Owasp Dpendency Checker

4.4.2.2 INTEGRACIÓN DE LAS HERRAMIENTAS EN JENKINS

4.4.2.2.1 JENKINS Y GITLAB. INTEGRACIÓN CONTINUA.

Para el Desarrollo de este proyecto se ha escogido un Código OpenSource vulnerable, muy utilizado en el mundo de la seguridad informática para conocer las distintas vulnerabilidades existentes, este Código pertenece a la aplicación **WebGoat,** una aplicación de prácticas de ciberseguridad. Por ello, empleando este código se simulará un desarrollo de una aplicación que, por cada nueva modificación en el código, se realizarán commits al repositorio Gitlab donde estará alojado.

El objetivo, consiste en ejecutar de forma automática el pipeline de Jenkins, cada vez que se detecte un push sobre el repositorio de la aplicación. De esta forma se ejecutará de forma automática un Sistema de despliegue continuo por cada cambio en la aplicación en cuestión, con las distintas etapas y pasos de seguridad, que compondrán el ciclo de vida de Desarrollo Seguro DevSecOps.

Para ello, en primer lugar, se va a generar un token, para poder interactuar con Gitlab de forma autenticada.

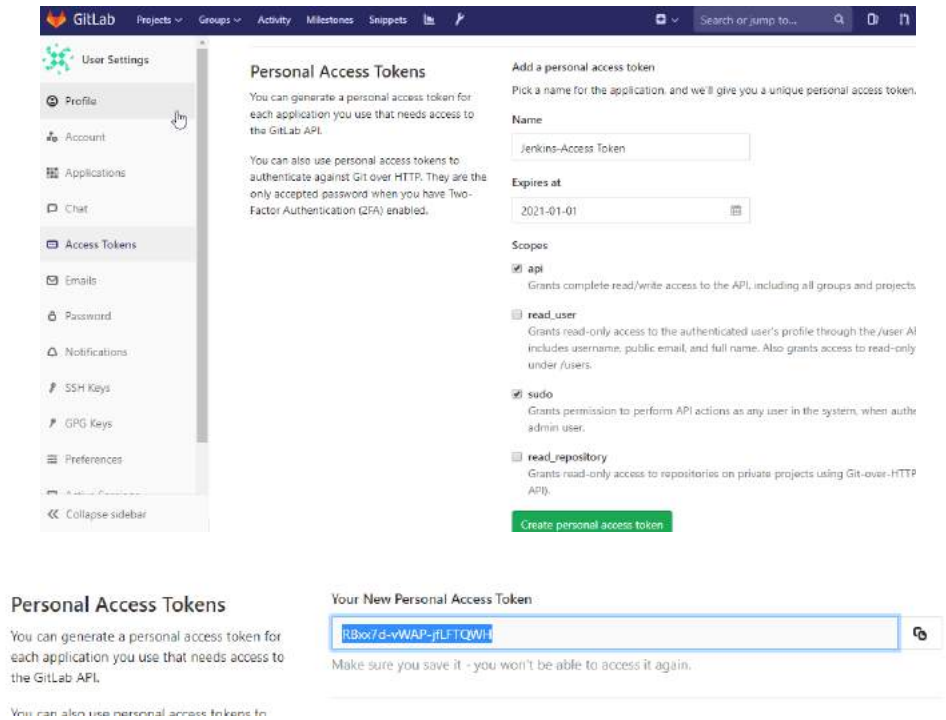

Ilustración 41: Generación token gitlab

Una vez generado el token, añadiremos las credenciales de Gitlab a la configuración de Jenkins:

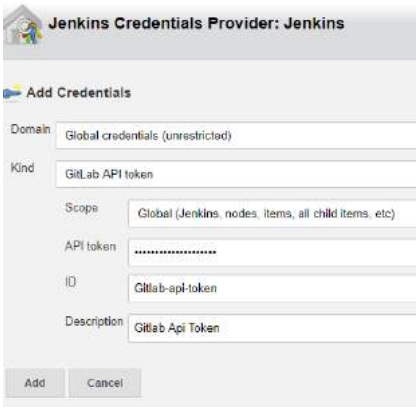

Ilustración 42: Añadir token de gitlab a Jenkins

A continuación, se configurará el plugin de Gitlab en Jenkins con las credenciales del token incluidas previamente, la url del repositorio y el nombre de conexión:

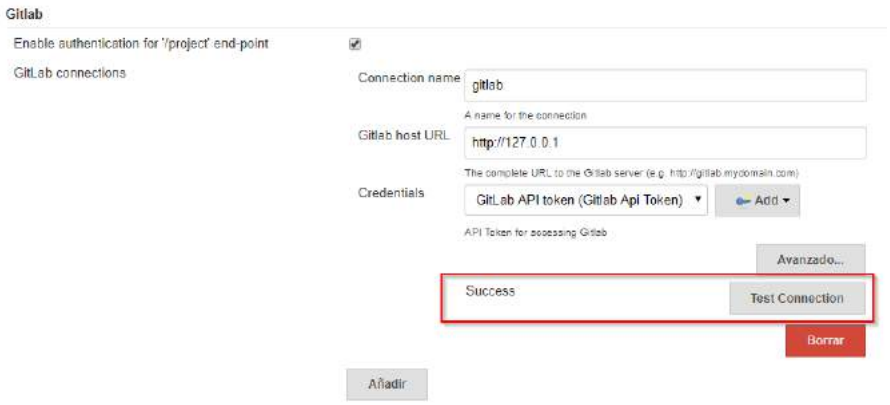

Ilustración 43: Configuración credenciales de gitlab en Jenkins

Como se puede observar, ya disponemos de conexión entre ambos sistemas.

El siguiente paso, es la creación de un nuevo proyecto de código, donde se realizará el desarrollo de la aplicación:

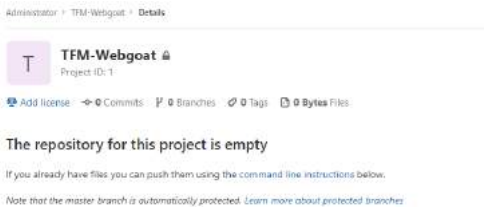

Ilustración 44: Creación repositorio

A su vez, se creará el job en Jenkins con el pipeline de ejecución. Además, este job dispondrá de un step del plugin de Gitlab, que será el encargado de monitorizar los cambios en el repositorio, cada vez que se realice un push sobre el mismo. Como se observa, este plugin genera una url, para poder ser integrada en un webhook de Gitlab, así como la posibilidad el tipo de eventos que ejecutarán el job:

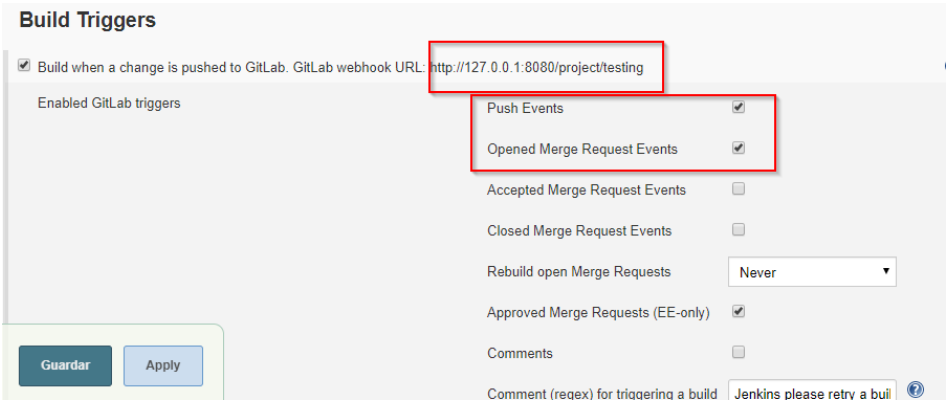

Ilustración 45: Configuración Job con repositorio

A continuación, en el repositorio en cuestión, debemos crear el Webhook que ejecutará el job cada vez que se detecte un push en el repositorio:

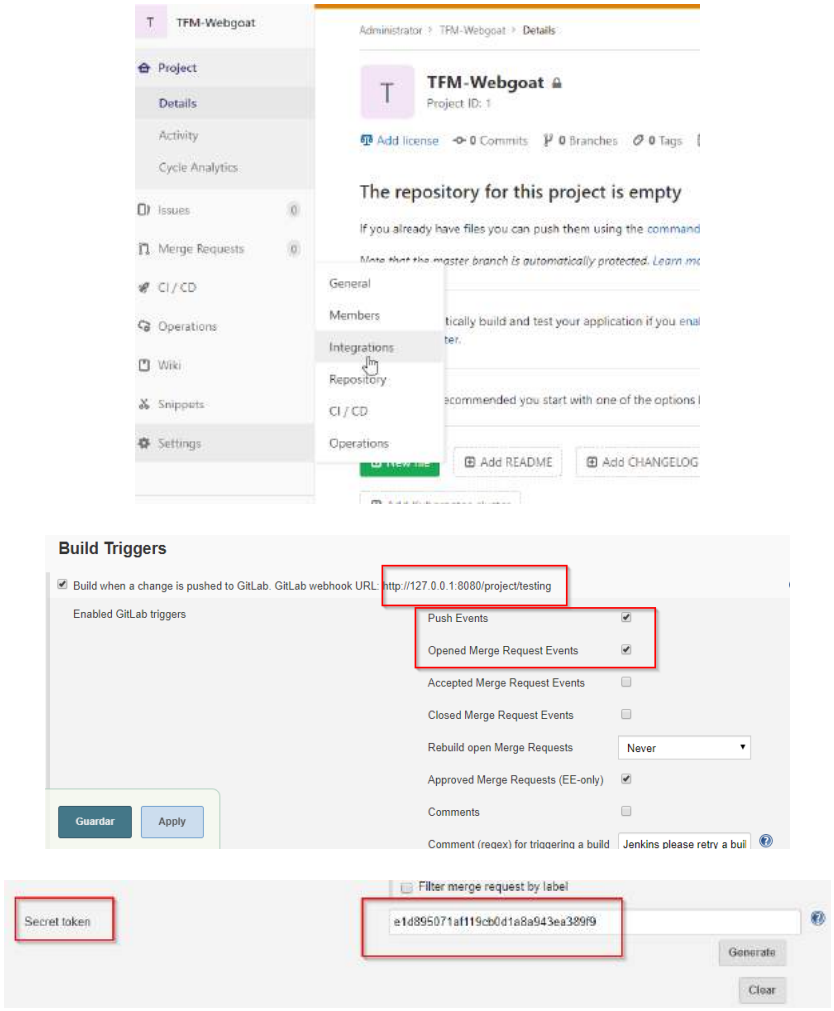

Ilustración 46: Integración Repositorio

Una vez creado el Webhook de integración con Jenkins, realizamos una prueba para comprobar que efectivamente el job se ejecuta al recibir un evento de tipo push.

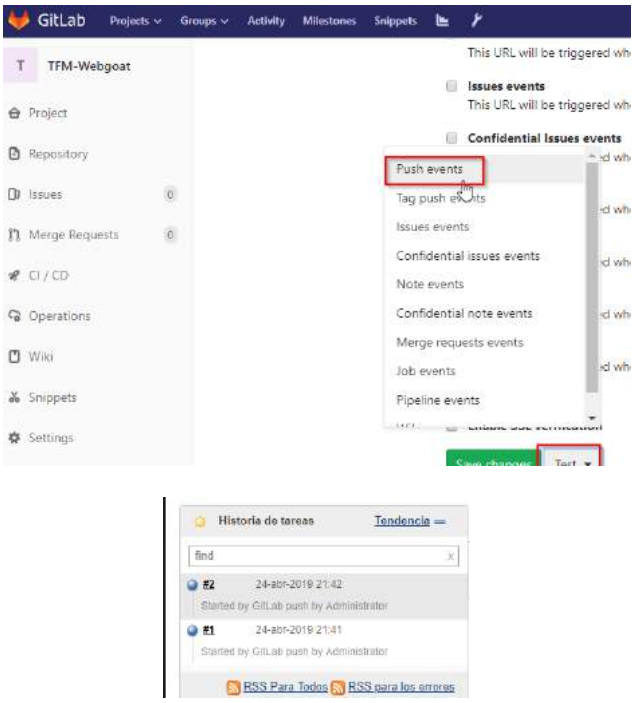

Ilustración 47: Prueba integración Jenkins-Gitlab

4.4.2.2.2 JENKINS Y SONARQUBE

Para la integración de Jenkins y SonarQube, es necesaria la instalación del plugin SonarQube Scanner.

En primer lugar, debemos generar el token de Sonar, para realizar las conexiones de manera segura y autenticada:

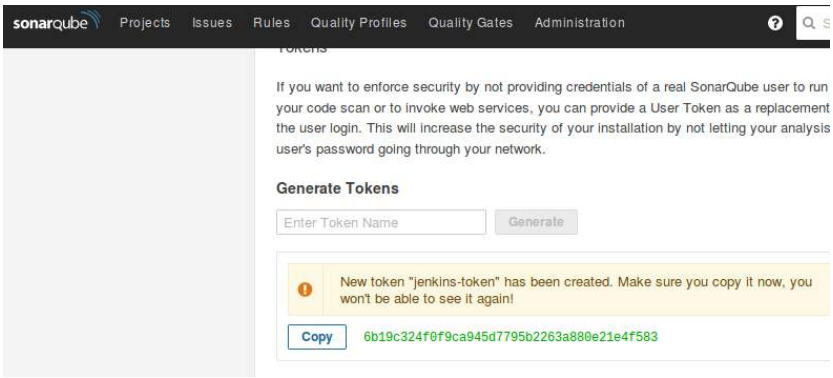

Ilustración 48: Token Sonarqube

En Jenkins, una vez generado el token, debemos configurar el servidor donde se encuentra Sonar, incluyendo el token recientemente generado:

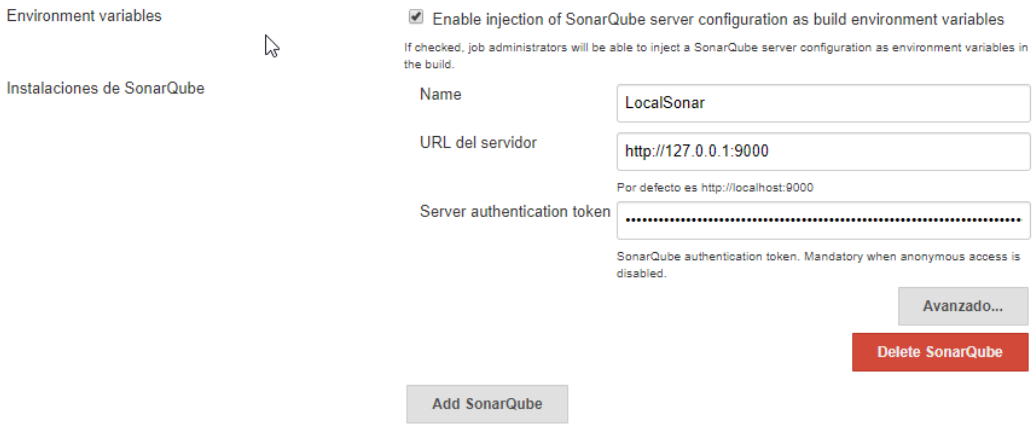

Ilustración 49: Configuración sonarqube en Jenkins

A continuación, debemos descargar el analizador local de SonarQube, que será ejecutado desde el pipeline de Jenkins:

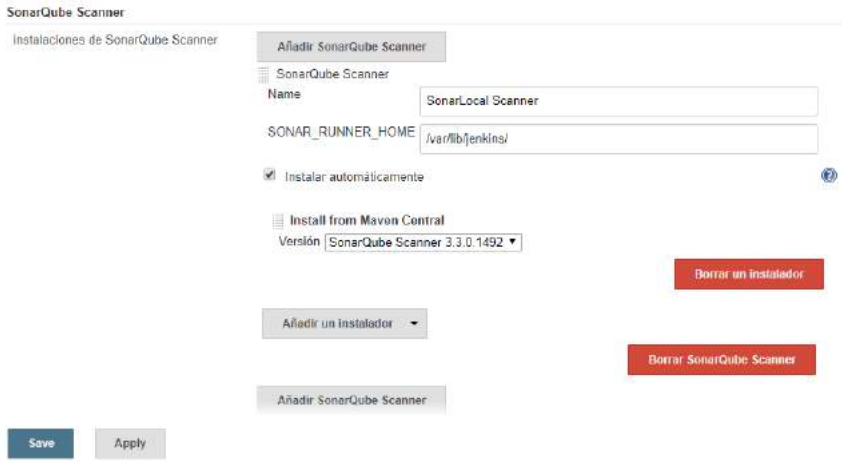

Ilustración 50: Analizador Local Sonarqube

4.4.2.3 DESARROLLO DEL PIPELINE DEVSECOPS

En primer lugar, para el desarrollo de este trabajo final de máster se ha utilizado un código Open Source de una aplicación vulnerable llamada WebGoat, este código se ha empleado para simular un proceso de integración continua en el proceso de desarrollo del software.

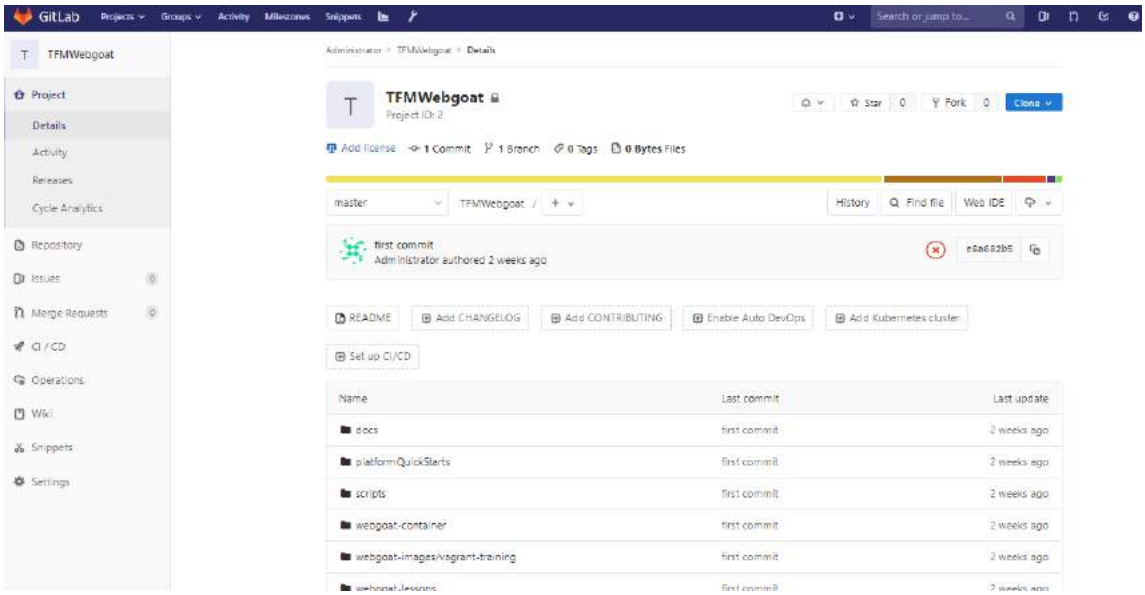

Ilustración 51: Código WebGoat y repositorio

De esta manera, cuando el equipo de desarrollo, realice un nuevo push en la rama "master" del repositorio http://127.0.0.1:81/root/TFMWebgoat.git, se ejecutará el webhook programado con anterioridad que disparará la ejecución de una nueva build del job de integración continua y despliegue continuo de la aplicación WebGoat.

Ilustración 52: Webhook

4.4.2.3.1 INTEGRACIÓN CONTINUA

Para el desarrollo e implementación del pipeline DevSecOps del ciclo de vida del desarrollo seguro del software, como se ha comentado se ha utilizado el repositorio con el código de la aplicación WebGoat alojado, y se ha configurado una integración con Jenkins cuando se detecten cambios en el código.

Para ello previamente se ha creado un nuevo job en Jenkins de tipo Pipeline Job:

Ilustración 53: Creación Pipeline Job

Su configuración, en primer lugar, debe disponer de un trigger que se ejecute al detectar un push en el código:

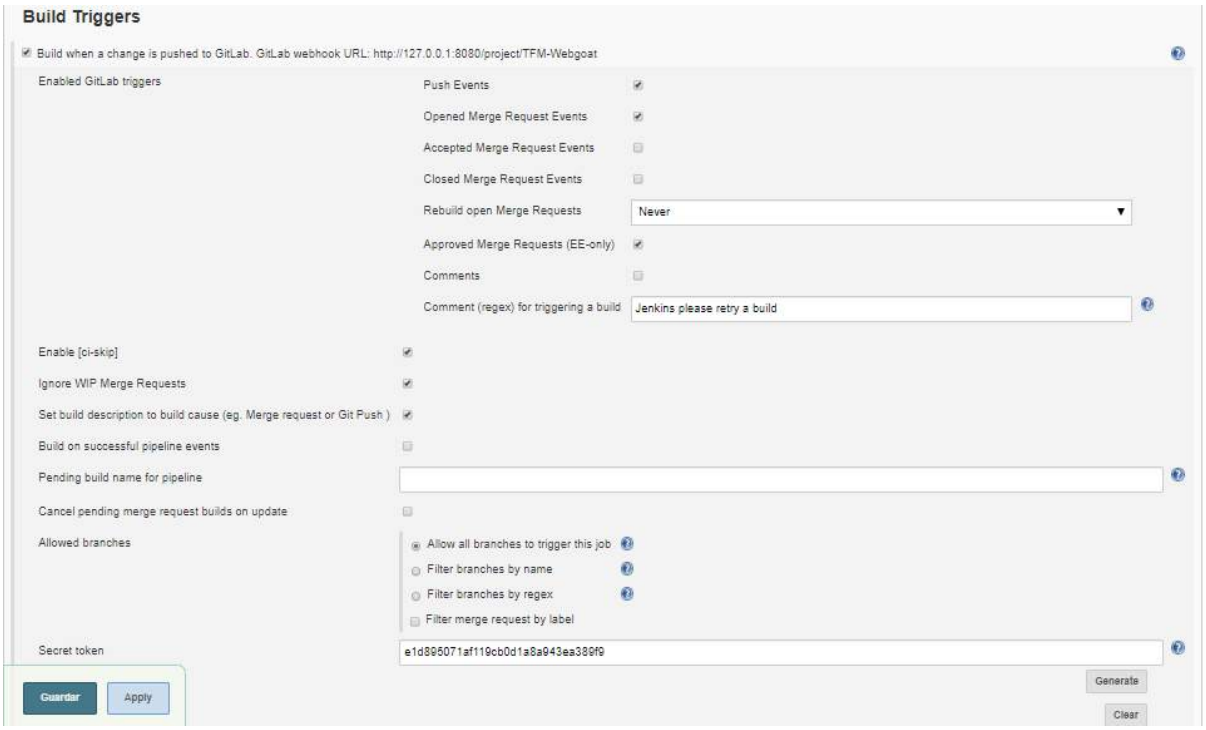

Ilustración 54: Configuración Pipeline Job

Como se observa, si se ejecutase una acción push o un evento merge en el código, se ejecutaría el job. Además, para dotar al sistema de más seguridad se ha generado un token, que ha sido incluido en el trigger de Gitlab. De esta forma logramos implantar un sistema de integración continua.

Ilustración 55: Pipeline

A continuación, se ha creado el pipeline en lenguaje Groovy, para ejecutar cada fase del ciclo de vida del software diseñado en puntos anteriores.

4.4.2.3.1.1 STAGE 1. MONITORIZACIÓN. CHECKOUT DEL REPOSITORIO

La primera fase de monitorización será la encargada tras ejecutarse el job por un push en el código, de descargarse la última versión del código alojado en el repositorio.

```
stages {
        stage('Checkout repository changes') {
            steps {
                echo 'Downloading the code from Gitlab'
                checkout([$class: 'GitSCM', branches: [[name: '*/master']],
                doGenerateSubmoduleConfigurations: false, extensions: [],
                submoduleCfg: [], userRemoteConfigs: [[credentialsId: 'gitlab-
user',
                url: 'http://127.0.0.1/root/TFMWebgoat']]])
            }
        }
```

Como se observa en el código, se realiza un checkout a la rama master, indicando las credenciales definidas en la fase de integración de las herramientas “gitlab-user” e indicando la url del repositorio.

4.4.2.3.1.2 STAGE 2. ANÁLISIS DE DEPENDENCIAS

Una vez disponemos en el Workspace del job de Jenkins la última versión del código, es decir la referente al último push, se ejecuta la fase del análisis de dependencias. Esta fase tiene definida la configuración de la herramienta Owasp Dependency Checker, que ha sido provista por el plugin “Owasp Dependency Checker plugin”

```
     stage('Dependency analysis') {
         steps {
            echo 'Performing Code Dependency analysis from the commit'
            dependencyCheckAnalyzer datadir: 'dependency-check-data',
            hintsFile: '', includeCsvReports: false,
includeHtmlReports: true,
            includeJsonReports: true, includeVulnReports: true,
            isAutoupdateDisabled: false, outdir: '.',
            scanpath: '', skipOnScmChange: false,
skipOnUpstreamChange: false, suppressionFile: '', zipExtensions: ''
            echo 'Finished!!!'
            dependencyCheckPublisher pattern: 'dependency-check-
report.xml'
            echo 'Publishing results...'

            }
        }
```

Este código, indica la ruta donde debe alojarse la base de datos con la información de las vulnerabilidades de Owasp y el tipo de reportes que queremos obtener. Este analizador recorrerá de forma recursiva todas las carpetas del workspace para identificar el archivo de gestión de dependencias, en este caso “pom.xml” ya que para compilar el proyecto se utiliza Maven.

Por último, se especifica al plugin dependencyCheckPublisher, que patrón debe buscar para identificar el reporte de resultados y poder visualizarlos directamente en Jenkins.

4.4.2.3.1.3 STAGE 3. ANÁLISIS ESTÁTICO DE LA SEGURIDAD DEL CÓDIGO SAST

Una vez se han analizado las dependencias de terceros, la siguiente fase consiste en el análisis estático de la seguridad del código fuente SAST, empleando SonarQube, que ya fue configurado y desplegado en fases anteriores. Por ello, en esta fase se debe especificar la URL donde se encuentra SonarQube, el nombre del proyecto que tendrá en Sonar, la versión, y alguna configuración adicional, como la exclusión de ficheros que no queramos analizar, y por último la ruta donde se encuentra el código.

```
        stage('SAST analysis') {
            steps {
                echo 'Performing SAST analysis to the code from the
commit'
                sh  "/var/lib/jenkins/sonar-scanner-3.3.0.1492-
linux/bin/sonar-scanner
                -Dsonar.host.url=http://127.0.0.1:9000
                -Dsonar.projectName=testing
                -Dsonar.projectVersion=1.0
                -Dsonar.projectKey=testing
                -Dsonar.exclusions=**/*.ts
                -Dsonar.sources=.
                -Dsonar.java.binaries=.
                -Dsonar.projectBaseDir=/var/lib/jenkins/workspace/TFM-
Webgoat"
                echo 'SAST Finished!!!'
            }
        }
```

4.4.2.3.2 BUILD. ENTREGA CONTINUA

A continuación, como hemos definido en la fase de diseño, queremos que nuestro ciclo de vida del software sea totalmente automático, desde la descarga del código y su análisis de seguridad, hasta la siguiente fase, disponer de un binario siempre disponible y funcional, es decir, un sistema de Entrega Continua CD.

4.4.2.3.2.1 STAGE 4. CONSTRUCCIÓN DE IMAGEN DOCKER

En esta fase, se realiza la construcción de la imagen Docker que posteriormente será desplegada en los diferentes entornos, para ello en primer lugar se compila el código empleando Maven, y a continuación, se ejecuta la imagen definida en el “Dockerfile”, que tiene como volumen el binario compilado producto del comando “mvn install”.

```
stage('Docker Build') {
            steps {
                echo 'Compiling code...'
                sh "mvn install -DskipTests"
                echo 'Jar file generated'
                dir ('./webgoat-server'){
                    echo 'Building docker image'
                    sh "docker build -t webgoat/webgoat-8.0 ."
                    echo 'Docker image builded'
                }
            }
        }
```

De esta manera, como resultado disponemos de una nueva imagen Docker, con él binario de la última versión pusheada al repositorio.

4.4.2.3.2.2 STAGE 5. ANÁLISIS DE LA SEGURIDAD DE LA IMAGEN DOCKER

Pero como se ha demostrado con frecuencia, las imágenes de Docker son puntos vulnerables, ya sea porque la imagen base y sus distintas capas contienen alguna vulnerabilidad, como alguna configuración realizada en el Dockerfile convierte la imagen en vulnerable. Por ello en esta fase, se ejecuta Clair juntamente con Klar, indicando la URL de la API de Clair, la ruta donde se ejecutará Klar y la imagen compilada para ser analizada.

```
stage('Docker security analysis') {
            steps {
                echo 'Performing Docker image analysis'
                sh "CLAIR_ADDR=localhost:6060 CLAIR_OUTPUT=High
CLAIR_THRESHOLD=100 /home/siriz/Escritorio/klar/bin/klar
webgoat/webgoat-8.0"
            }
        }
```

4.4.2.3.3 DEPLOY. DESPLIEGUE CONTINUO

Una vez ya dispongo de una imagen de Docker lista para ser desplegada y analizada, comienzan las fases de despliegue, de forma totalmente automática, definiendo así el sistema como un sistema de Despliegue continuo CD.

4.4.2.3.3.1 STAGE 6. DESPLIEGUE DE LA APLICACIÓN

El objetivo de esta fase es el despliegue de la aplicación, por ello en primer lugar, se sube la imagen de Docker creada en la anterior fase al repositorio de Docker "Docker Registry" versionándola con un tag. A continuación, se arranca el contenedor a partir de la imagen y se despliega la aplicación en el puerto 8080, para este caso ha sido llamada "vulnerable-application".

```
stage('Docker Deployment') {
            steps {
                echo 'Deploying the application'
                sh "docker tag webgoat/webgoat-8.0
localhost:4443/webgoat-8.0"
                sh "docker push localhost:4443/webgoat-8.0"
                sh "docker run -d --name vulnerable-application -p
8888:8080 localhost:4443/webgoat-8.0"
            }
        }
```

4.4.2.3.3.2 STAGE 7. ANÁLISIS DINÁMICO DAST

Una vez se ha desplegado la aplicación en el puerto indicado, se procede a realizar el análisis Dinámico DAST utilizando ZAP.

En primer lugar, se especifica la URL donde se ejecutará el Proxy ZAP y su puerto, y la ubicación del binario de ZAP. El siguiente paso es verificar que efectivamente tenemos conexión con la aplicación a través del proxy, y por último se ejecutará el DAST, primero en modo Crawler y posteriormente con la política en modo ataque. Los resultados se visualizarán a través del propio Job de Jenkins.

```
stage('DAST Analysis') {
            steps {
                script {
                    startZap(host: "127.0.0.1", port: 9091,
timeout:500, zapHome: "/opt/ZAP_2.7.0", allowedHosts:['127.0.0.1'])
                    sh "mvn verify -Dhttp.proxyHost=127.0.0.1 -
Dhttp.proxyPort=9091 -Dhttps.proxyHost=127.0.0.1 -
Dhttps.proxyPort=9091 -DskipTests"
                    runZapCrawler(host:
"http://localhost:8888/WebGoat")
                    runZapAttack()
                }

            }
        }
```

4.4.2.3.4 POST BUILDS

Por último, para finalizar esta iteración del ciclo de vida del software, se ejecutan una serie de PostBuilds. El primero de ellos, consiste en la creación del fichero de resultados del análisis Dinámico para ser visualizados en Jenkins.

Por último, se eliminará y parará el contenedor con la aplicación desplegada.

```
    post {
        always {
            script {
                sh "touch zapFalsePositives.json"
                archiveZap(failAllAlerts: 100, failHighAlerts: 0,
failMediumAlerts: 0, failLowAlerts: 0, falsePositivesFilePath:
"zapFalsePositives.json")
                sh "docker rm -f vulnerable-application"

            }
        }
    }
}
```

5. SOLUCIÓN FINAL DEVSECOPS

Con el resultado final de este pipeline, la fase de especificación de requisitos, la implantación, configuración y despliegue de todas las herramientas involucradas, se ha logrado ejecutar el ciclo completo **DevSecOps** abordando todas las fases del ciclo de vida del desarrollo del software, de forma totalmente **automatizada** desde una perspectiva de **Despliegue Continuo**, securizando cada una de las fases con diversas herramientas e integrando todo el proceso bajo una sola, **centralizando** tanto los logs de ejecución, como los resultados obtenidos, todo ello de una manera ágil y logrando un **feedback continuo de cada una de las fases.**

- **Definición de requisitos de seguridad y threat modelling**

Ilustración 56: Requisitos de seguridad

- **Pipeline DevSecOps.**

Ilustración 57: Ejecución correcta Pipeline

5.1 RESULTADOS OBTENIDOS

STAGE 1. MONITORIZACIÓN. CHECKOUT DEL REPOSITORIO

Tras realizarse un push en el repositorio, podemos comprobar en los logs de ejecución, como el sistema ha comprobado los cambios en el repositorio y se ha realizado un checkout del repositorio:

```
[Pipeline] stage
[Pipeline] { (Checkout repository changes)
[Pipeline] echo
Downloading the code from Gitlab
[Pipeline] checkout
using credential gitlab-user
Cloning the remote Git repository
Cloning repository http://127.0.0.1/root/TFMWebgoat
 > git init /var/lib/jenkins/workspace/TFM-Webgoat # timeout=10
Fetching upstream changes from http://127.0.0.1/root/TFMWebgoat
 > git --version # timeout=10
using GIT_ASKPASS to set credentials credenciales para conectar con gitlab
 > git fetch --tags --progress http://127.0.0.1/root/TFMWebgoat +refs/heads/*:refs/remotes/origin/*
 > git config remote.origin.url http://127.0.0.1/root/TFMWebgoat # timeout=10
 > git config --add remote.origin.fetch +refs/heads/*:refs/remotes/origin/* # timeout=10
 > git config remote.origin.url http://127.0.0.1/root/TFMWebgoat # timeout=10
Fetching upstream changes from http://127.0.0.1/root/TFMWebgoat
using GIT_ASKPASS to set credentials credenciales para conectar con gitlab
 > git fetch --tags --progress http://127.0.0.1/root/TFMWebgoat +refs/heads/*:refs/remotes/origin/*
 > git rev-parse refs/remotes/origin/master^{commit} # timeout=10
 > git rev-parse refs/remotes/origin/origin/master^{commit} # timeout=10
Checking out Revision e8a682b5c6f4da799fba67f95e9f4ae7abb4cbb3 (refs/remotes/origin/master)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f e8a682b5c6f4da799fba67f95e9f4ae7abb4cbb3
Commit message: "first commit"
```

STAGE 2. ANÁLISIS DE DEPENDENCIAS

En este momento se ejecuta el análisis de dependencias:

```
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Dependency analysis)
[Pipeline] echo
Performing Code Dependency analysis from the commit
[Pipeline] dependencyCheckAnalyzer
[DependencyCheck] OWASP Dependency-Check Plugin v4.0.2
[DependencyCheck] Executing Dependency-Check with the following options:
[DependencyCheck]   -name = TFM-Webgoat
[DependencyCheck]   -scanPath = /var/lib/jenkins/workspace/TFM-Webgoat
[DependencyCheck]   -outputDirectory = /var/lib/jenkins/workspace/TFM-Webgoat
[DependencyCheck]   -dataDirectory = /var/lib/jenkins/workspace/TFM-Webgoat/dependency-check-data
[DependencyCheck]   -dataMirroringType = none
Finished!!!
[Pipeline] dependencyCheckPublisher
[DependencyCheck] Collecting Dependency-Check analysis files...
[DependencyCheck] Searching for all files in /var/lib/jenkins/workspace/TFM-Webgoat that match the pattern dependency-check-report.xml
[DependencyCheck] Parsing 1 file in /var/lib/jenkins/workspace/TFM-Webgoat
[DependencyCheck] Successfully parsed file /var/lib/jenkins/workspace/TFM-Webgoat/dependency-check-report.xml with 25 unique warnings and 0 duplicates.
Skipping warnings blame since pipelines do not have an SCM link.%n
[Pipeline] echo
Publishing results...
```

Como se observa en la siguiente imagen, se publican los resultados en el propio job de Jenkins. Esta aplicación emplea dependencias vulnerables, 20 media y 5 bajas:

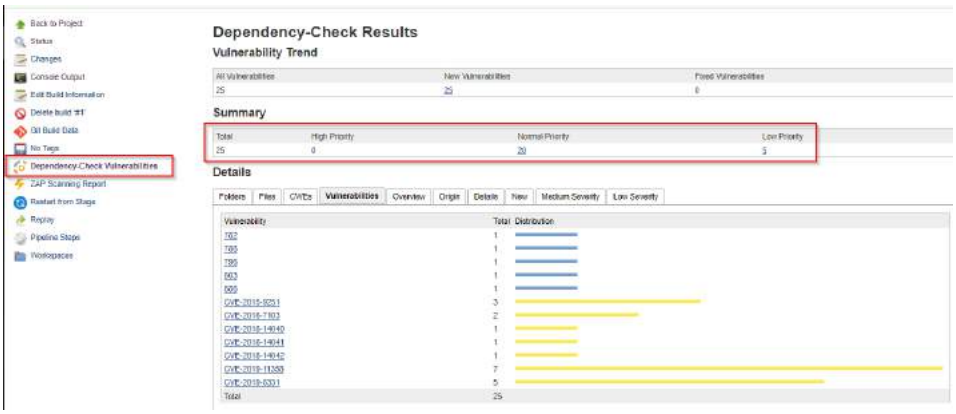

Ilustración 58: Resultados Dependency Check

Dependency Check nos indica bastante información acerca de las vulnerabilidades, desde el CWE o CVE que identifica la vulnerabilidad, a la librería y la vulnerabilidad a la que es vulnerable:

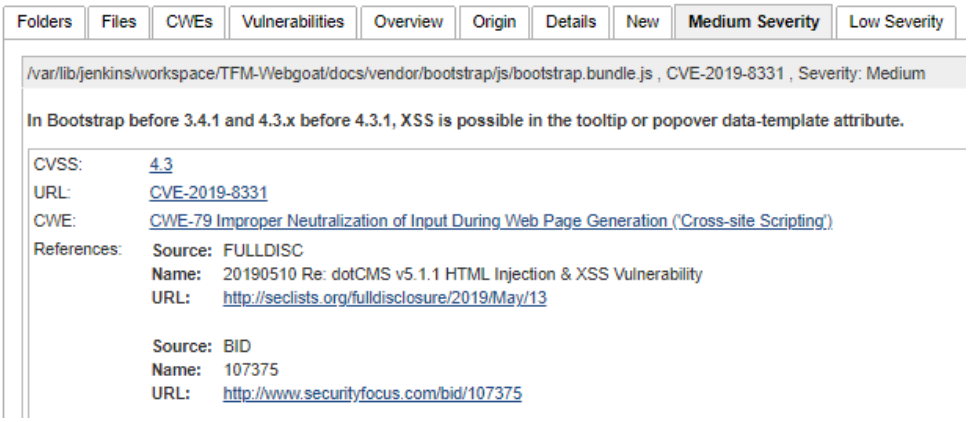

Ilustración 59: Vulnerabilidad Dependency check

STAGE 3. ANÁLISIS ESTÁTICO DE LA SEGURIDAD DEL CÓDIGO SAST

El siguiente paso, ejecuta el análisis estático publicando los resultados en la propia interfaz de SonarQube, según la configuración definida en el pipeline:

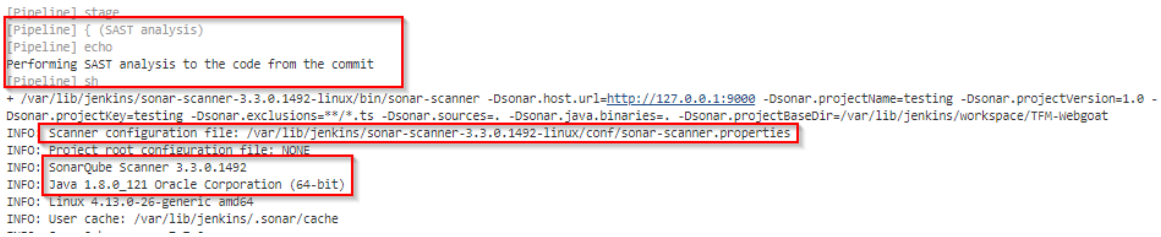

Ilustración 60: Resultados SonarQube

Una vez ha finalizado este step, ya disponemos del proyecto en SonarQube con los resultados obtenidos del análisis:

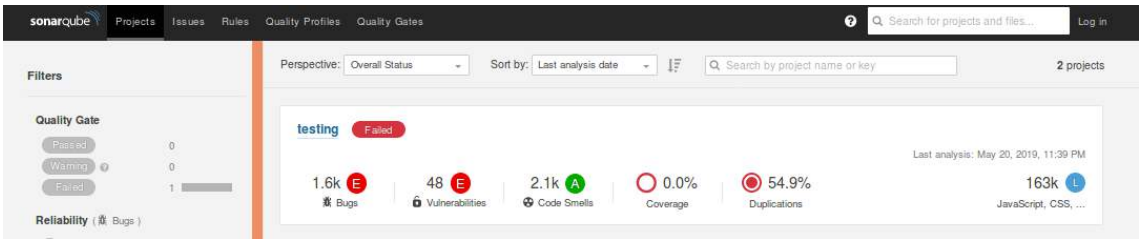

Ilustración 61: Resultados Sonarqube

Sonar aporta bastante información valiosa, como el número de líneas de código, lenguajes detectados, y una vista resumen de las vulnerabilidades detectadas categorizadas por defectos de seguridad (vulnerabilidades) o defectos de la calidad o eficiencia del código(bugs):

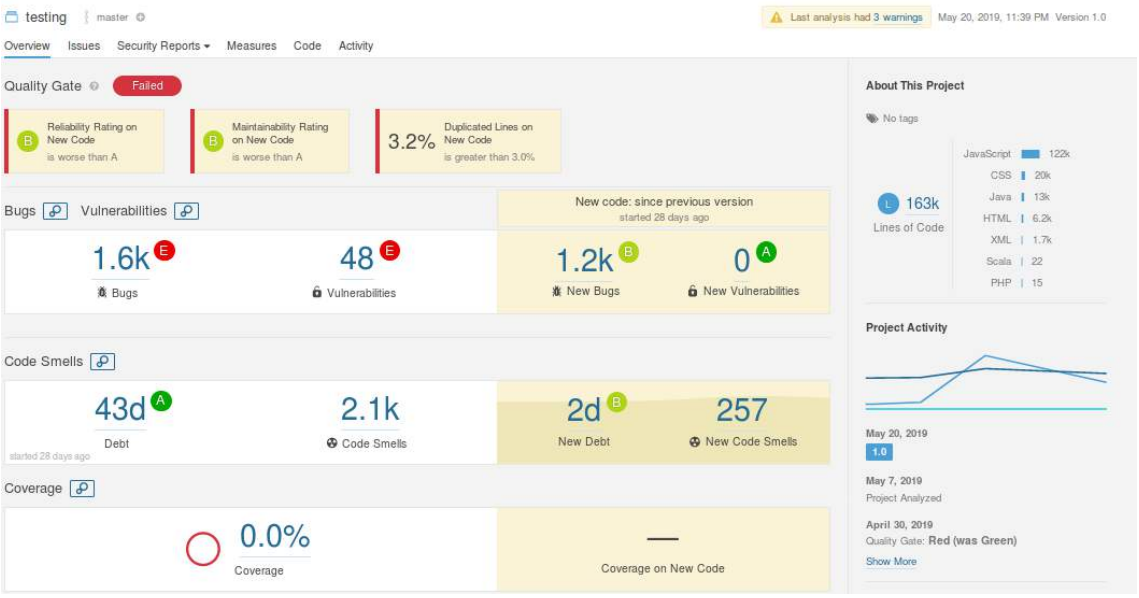

Ilustración 62: Detalle resultados Sonarqube

Esta aplicación tiene un número elevado de bugs lo que nos indica que no está correctamente desarrollada desde el punto de vista de la calidad del código. Pero los elementos de mayor riesgo detectados son las 48 vulnerabilidades. Si accedemos a la vista de vulnerabilidades podremos ver que se han detectado vulnerabilidades de riesgo bloqueante, por lo que no sería recomendada una subida al entorno de producción. Un ejemplo de la vista donde se identifica el punto vulnerable, y el flujo de la vulnerabilidad es el siguiente:

Ilustración 63: Hallazgo Sonarqube

STAGE 4. CONSTRUCCIÓN DE IMAGEN DOCKER

En esta fase, se muestra como en primer lugar se compila el código y posteriormente se genera la imagen de Docker, incluyendo el binario resultante:

```
[Pipeline] stage
[Pipeline] { (Docker Build)
[Pipeline] echo
Compiling code...
[Pipeline] sh
+ mvn install -DskipTests
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by com.google.inject.internal.cglib.core.$ReflectUtils$1 (file:/usr/share/ma
java.lang.ClassLoader.defineClass(java.lang.String,byte[],int,int,java.security.ProtectionDomain)
WARNING: Please consider reporting this to the maintainers of com.google.inject.internal.cglib.core.$ReflectUti
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------

Jar file generated
[Pipeline] dir
Running in /var/lib/jenkins/workspace/TFM-Webgoat/webgoat-server
[Pipeline] {
[Pipeline] echo
Building docker image
[Pipeline] sh
+ docker build -t webgoat/webgoat-8.0 .
Sending build context to Docker daemon  82.69MB

Step 1/9 : FROM openjdk:11.0.1-jre-slim-stretch
 ---> 49b31a72a85a
Step 2/9 : ARG webgoat_version=v8.0.0-SNAPSHOT
 ---> Using cache
 ---> f3900240ae17
Step 3/9 : RUN   apt-get update && apt-get install &&   useradd --home-dir /home/webgoat --create-home -U webgoat
 ---> Using cache
 ---> 97d380f0da50
Step 4/9 : USER webgoat
 ---> Using cache
 ---> bbf036bff648
Step 5/9 : RUN cd /home/webgoat/; mkdir -p .webgoat-${webgoat_version}
 ---> Using cache
 ---> 8a8ebfdc6184
Step 6/9 : COPY target/webgoat-server-${webgoat_version}.jar /home/webgoat/webgoat.jar
 ---> 440a85006c32
Step 7/9 : EXPOSE 8080
 ---> Running in 73f84c0029a3
Removing intermediate container 73f84c0029a3
 ---> bd5ad92411d6
Step 8/9 : ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/./urandom", "-jar", "/home/webgoat/webgoat.jar"]
 ---> Running in 8619224f8c6a
Removing intermediate container 8619224f8c6a
 ---> 81930b5b7bc1
Step 9/9 : CMD ["--server.port=8080", "--server.address=0.0.0.0"]
 ---> Running in 6bae7870753f
Removing intermediate container 6bae7870753f
 ---> c97027106885e
Successfully built c97027106885e
Successfully tagged webgoat/webgoat-8.0:latest
```

STAGE 5. ANÁLISIS DE LA SEGURIDAD DE LA IMAGEN DOCKER

Una vez creada la imagen, es el turno de comprobar su seguridad empleando Clair y la Klar que permite ejecutar un análisis de Clair a través del terminal:

```
[Pipeline] stage
[Pipeline] { (Docker security analysis)
[Pipeline] echo
Performing Docker image analysis
[Pipeline] sh
+ CLAIR_ADDR=localhost:6060 CLAIR_OUTPUT=High CLAIR_THRESHOLD=100 /home/siriz/Escritorio/klar/bin/klar webgoat/webgoat-8.0
Clair timeout 1m0s
docker timeout: 1m0s
no whitelist file
Analysing 9 layers
Got results from Clair API v1
Found 104 vulnerabilities
Unknown: 4
Negligible: 40
Low: 19
Medium: 27
High: 14
```

Los resultados nos indican que se trata de una imagen muy vulnerable con 14 vulnerabilidades altas en sus layers, 27 medias y 19 bajas, la información que nos indica Clair, es el CVE, layer afectada y una descripción de la vulnerabilidad. Esta imagen no debería subirse al repositorio de Docker, Docker Registry ya que es vulnerable:

```
CVE-2018-15686: [High]
Found in: systemd [232-25+deb9u8]
Fixed By: 232-25+deb9u10
A vulnerability in unit_deserialize of systemd allows an attacker to supply arbitrary state across systemd re-execution via Noti
influence systemd execution and possibly lead to root privilege escalation. Affected releases are systemd versions up to and inc
https://security-tracker.debian.org/tracker/CVE-2018-15686
----------------------------------------
CVE-2017-12424: [High]
Found in: shadow [1:4.4-4.1]
Fixed By:
In shadow before 4.5, the newusers tool could be made to manipulate internal data structures in ways unintended by the authors.
buffer overflow or other memory corruption) or other unspecified behaviors. This crosses a privilege boundary in, for example, c
Control Panel allows an unprivileged user account to create subaccounts.
https://security-tracker.debian.org/tracker/CVE-2017-12424
----------------------------------------
CVE-2016-2779: [High]
Found in: util-linux [2.29.2-1+deb9u1]
Fixed By:
runuser in util-linux allows local users to escape to the parent session via a crafted TIOCSTI ioctl call, which pushes characte
https://security-tracker.debian.org/tracker/CVE-2016-2779
----------------------------------------
```

STAGE 6. DESPLIEGUE DE LA APLICACIÓN

En esta fase, se subirá la imagen al Docker Registry y posteriormente se desplegará la aplicación:

```
[Pipeline] stage
[Pipeline] { (Docker Deployment)
[Pipeline] echo
Deploying the application
[Pipeline] sh
+ docker tag webgoat/webgoat-8.0 localhost:4443/webgoat-8.0
[Pipeline] sh
+ docker push localhost:4443/webgoat-8.0
The push refers to repository [localhost:4443/webgoat-8.0]
388282b71f87: Layer already exists
a2759aca23c6: Pushed
latest: digest: sha256:24d3089752bce3040bd9b427bb8e5546dbaeebf146cd21c1b52679d1eafb9607 size: 2203
[Pipeline] sh
+ docker run -d --name vulnerable-application -p 8888:8080 localhost:4443/webgoat-8.0
7a313b530ef516e82bb2da5c46e25ef49d11366d8057340b658d764e30f272ef
```

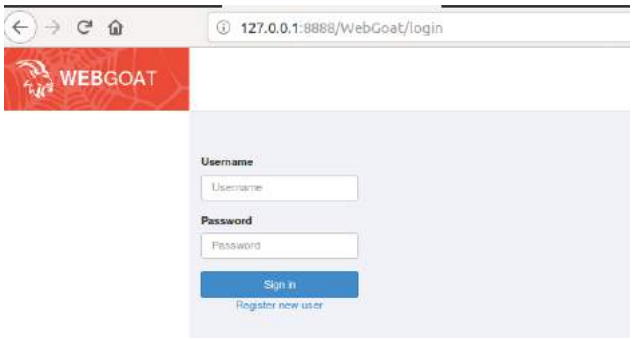

Ilustración 64: WebGoat

STAGE 7. ANÁLISIS DINÁMICO DAST

La última fase, una vez ya disponemos de la aplicación desplegada es la ejecución del análisis Dinámico del código de la aplicación DAST empleando ZAP:

```
[Pipeline] stage
[Pipeline] { (DAST Analysis)
[Pipeline] script
[Pipeline] {
[Pipeline] startZap
zap: Starting ZAP on port 9091...
[TFM-Webgoat] $ /opt/ZAP_2.7.0/zap.sh -daemon -host 127.0.0.1 -port 9091 -config api.disablekey=true -c
onfig connection.timeoutInSecs=600
[Pipeline] runZapCrawler
zap: Starting crawler on host http://localhost:8888/WebGoat...
[Pipeline] runZapAttack
zap: Starting attack...
zap: Set mode to attack mode
[Pipeline] }
```

Una vez ha finalizado el análisis DAST, disponemos de los resultados en Jenkins gracias al paso de las Post Actions, que parsea el json de resultados y convierte en formato interpretable por Jenkins:

Ilustración 65: Acceso a resultados ZAP

El resultado de la ejecución del DAST han sido 4 vulnerabilidades de criticidad baja:

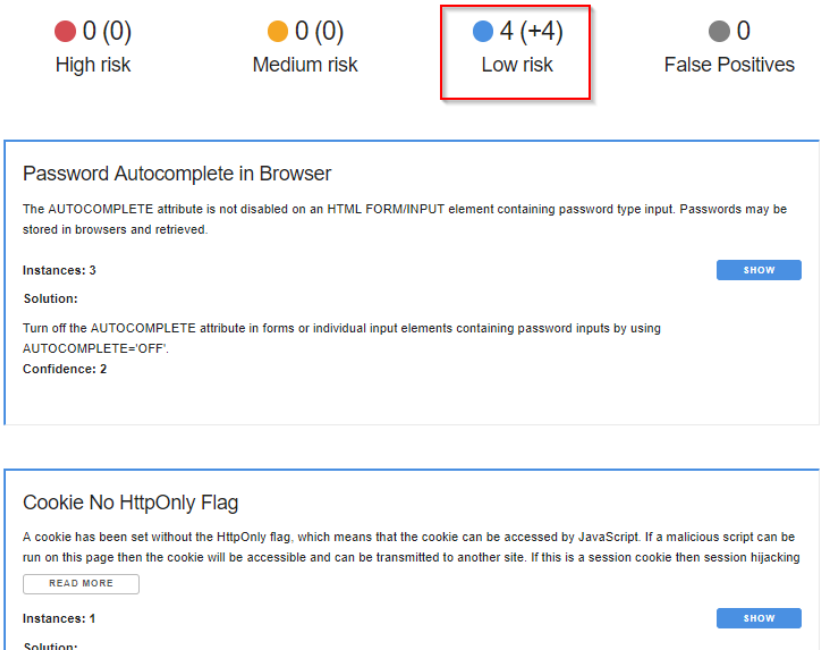

Ilustración 66: Resultados ZAP

6. CONCLUSIONES

6.1 CONCLUSIONES Y OBJETIVOS CUMPLIDOS

Como hemos ido viendo a lo largo del desarrollo de este trabajo de fin de máster, la seguridad a pesar de ser una pieza clave para la integridad, confidencialidad y reputación de una organización, se deja siempre para las últimas fases del ciclo de vida del software debido a la idea de que es algo limitante, que retrasa los tiempos de entrega, y es una labor de los especialistas en seguridad.

En este trabajo se ha demostrado la importancia de añadir la seguridad en todas las fases y se ha eliminado la idea de que seguridad y desarrollo son incompatibles, como que la seguridad solo debe incluirse al final del ciclo.

Finalmente se ha logrado realizar un diseño e implementación de un ciclo de vida del desarrollo del software seguro, gracias a la cultura DevSecOps.

- Diseño de un ciclo de vida del software seguro.

Se han definido en primer lugar unos requisitos de seguridad desde la fase de diseño, estudiando las posibles implicaciones de seguridad que puede tener una aplicación desarrollada por una organización, las diferentes conexiones e interacciones entre sistemas que puede incluir, la necesidad de confidencialidad de la información que se gestiona y la importancia del correcto control de errores evitando errores no contemplados de ejecución y exposiciones de información sensible.

- Análisis y definición de políticas y requisitos de seguridad que debe cumplir el software.

Por otor lado, se ha llevado a cabo el desarrollo de un pipeline, que incluye todas las fases que deben pasar desde que se está desarrollando una aplicación y se libera una nueva versión hasta su compilación y despliegue. Este proceso es totalmente iterativo y no existirá ni un solo cambio en el repositorio que no se haya comprobado su repercusión en seguridad en las distintas fases del SDLC.

Esto se ha realizado de forma 100% automática sin requerir de la interacción humana manual para realizar el checkout del código, compilar y desplegar, así como la ejecución de todas las pruebas de seguridad.

- Implementación de un sistema de despliegue continuo.
- Pruebas de seguridad en cada una de las fases del ciclo de vida del software.

Por lo tanto, se ha implementado un sistema que teniendo en cuenta los requisitos de seguridad recabados tanto para el desarrollo como para la implementación de las

políticas a aplicar de seguridad, incluya diversas pruebas de seguridad SAST, DAST, análisis de dependencias, análisis de imágenes docker que nos asegura que la versión que ha sido desplegada ha sido validada en todas sus fases y se conocen los riesgos de seguridad que contiene y que implicaría una salida de esa aplicación al mundo.

- Integración de las políticas de seguridad y el sistema de despliegue continuo en el ciclo de vida DevSecOps
- **Definición e implantación de un ciclo de vida del desarrollo del software seguro DevSecOps**

TABLA DE OBJETIVOS CUMPLIDOS
Objetivo Principal
➢ **Definición e implantación de un ciclo de vida del desarrollo del software seguro DevSecOps**
Objetivos secundarios
➢ **Diseño de un ciclo de vida del software seguro.** ➢ **Análisis y definición de políticas y requisitos de seguridad que debe cumplir el software.** ➢ **Implementación de un sistema de despliegue continuo.** ➢ **Pruebas de seguridad en cada una de las fases del ciclo de vida del software.** ➢ **Integración de las políticas de seguridad y el sistema de despliegue continuo en el ciclo de vida DevSecOps**

Tabla 27: Objetivos Cumplidos

BIBLIOGRAFÍA

Aiello, B. (2013). *cmcrossroads.* Obtenido de https://www.cmcrossroads.com/article/using-alm-drive-devops

Avansis. (11 de 04 de 2018). Obtenido de https://www.avansis.es/2018/04/11/1669/

Bienvenido, Á. (28 de 09 de 2018). *kabel.* Obtenido de https://www.kabel.es/iniciacion-ciclo-vida-devops/

consultor-it. (10 de 01 de 2016). Obtenido de https://www.consultor-it.com/articulo/70107/application-lifecycle-management-alm/otros/guia-de-software-application-lifecycle-management-alm

Drinkwater, D. (09 de 01 de 2019). *csoonline.* Obtenido de https://www.csoonline.com/article/3245748/what-is-devsecops-developing-more-secure-applications.html

Gartner. (2019). *www.gartner.com.* Obtenido de https://www.gartner.com/it-glossary/static-application-security-testing-sast

Ghahrai, A. (12 de 12 de 2018). *testingexcellence.* Obtenido de https://www.testingexcellence.com/software-development-life-cycle-sdlc-phases/

gitlab. (2019). Obtenido de https://docs.gitlab.com/ee/user/project/merge_requests/sast.html

gmendez. (22 de 10 de 2018). *fdi.* Obtenido de https://www.fdi.ucm.es/profesor/gmendez/docs/is0809/ieee830.pdf

Owasp. (2018). *owaspsamm.* Obtenido de https://owaspsamm.org/

Roberth G. Figueroa1, C. J. (Febrero de 2007). *researchgate*. Obtenido de https://www.researchgate.net/publication/299506242_METODOLOGIAS_TRADICIONALES_VS_METODOLOGIAS_AGILES

Saavedra, F. (29 de 05 de 2018). *gestiopolis*. Obtenido de Seguridad en SDLC. Ciclo de Vida de Desarrollo de Software: https://www.gestiopolis.com/seguridad-en-sdlc-ciclo-de-vida-de-desarrollo-de-software/

School, O. B. (s.f.). *obs-edu*. Obtenido de Qué son las metodologías de desarrollo de software: https://www.obs-edu.com/es/blog-project-management/metodologia-agile/que-son-las-metodologias-de-desarrollo-de-software

Veracode, C. (15 de 02 de 2018). Obtenido de https://www.itdigitalsecurity.es/actualidad/2018/02/la-integracion-de-la-seguridad-en-el-desarrollo-de-software-entre-el-reto-y-la-necesidad

7. ANEXOS

7.1 JENKINSFILE

```
pipeline {
  agent any
  environment {
    //Variables de entorno
    workspace = pwd()
  }
  stages {
    stage('Checkout repository changes') {
      steps {
        echo 'Downloading the code from Gitlab'
        checkout([$class: 'GitSCM', branches: [[name: '*/master']],
        doGenerateSubmoduleConfigurations: false, extensions: [],
        submoduleCfg: [], userRemoteConfigs: [[credentialsId: 'gitlab-user',
        url: 'http://127.0.0.1/root/TFMWebgoat']]])
      }
    }
    stage('Dependency analysis') {
      steps {
        echo 'Performing Code Dependency analysis from the commit'
        dependencyCheckAnalyzer datadir: 'dependency-check-data',
        hintsFile: '', includeCsvReports: false, includeHtmlReports: true,
        includeJsonReports: true, includeVulnReports: true,
        isAutoupdateDisabled: false, outdir: '.',
        scanpath: '', skipOnScmChange: false, skipOnUpstreamChange: false, suppressionFile: '', zipExtensions: ''
        echo 'Finished!!!'
        dependencyCheckPublisher pattern: 'dependency-check-report.xml'
        echo 'Publishing results...'
      }
    }
    stage('SAST analysis') {
      steps {
        echo 'Performing SAST analysis to the code from the commit'
                                    sh  "/var/lib/jenkins/sonar-scanner-3.3.0.1492-linux/bin/sonar-scanner -Dsonar.host.url=http://127.0.0.1:9000 -
            Dsonar.projectName=testing -Dsonar.projectVersion=1.0 -Dsonar.projectKey=testing -Dsonar.exclusions=**/*.ts  -Dsonar.sources=. -Dsonar.java.binaries=. -
            Dsonar.projectBaseDir=/var/lib/jenkins/workspace/TFM-Webgoat"
        echo 'SAST Finished!!!'
      }
    }
    stage('Docker Build') {
      steps {
        echo 'Compiling code...'
        sh "mvn install -DskipTests"
        echo 'Jar file generated'
        dir ('./webgoat-server'){
          echo 'Building docker image'
          sh "docker build -t webgoat/webgoat-8.0 ."
          echo 'Docker image builded'
        }
      }
    }
    stage('Docker security analysis') {
      steps {
        echo 'Performing Docker image analysis'
        sh "CLAIR_ADDR=localhost:6060 CLAIR_OUTPUT=High CLAIR_THRESHOLD=100 /home/siriz/Escritorio/klar/bin/klar webgoat/webgoat-8.0"
      }
    }
    stage('Docker Deployment') {
      steps {
        echo 'Deploying the application'
        sh "docker tag webgoat/webgoat-8.0 localhost:4443/webgoat-8.0"
        sh "docker push localhost:4443/webgoat-8.0"
        sh "docker run -d --name vulnerable-application -p 8888:8080 localhost:4443/webgoat-8.0"
      }
    }
    stage('DAST Analysis') {
      steps {
        script {
          startZap(host: "127.0.0.1", port: 9091, timeout:500, zapHome: "/opt/ZAP_2.7.0", allowedHosts:['127.0.0.1'])
          sh "mvn verify -Dhttp.proxyHost=127.0.0.1 -Dhttp.proxyPort=9091 -Dhttps.proxyHost=127.0.0.1 -Dhttps.proxyPort=9091 -DskipTests"
          runZapCrawler(host: "http://localhost:8888/WebGoat")
          runZapAttack()
        }

      }
    }
  }
  post {
    always {
      script {
        sh "touch zapFalsePositives.json"
        archiveZap(failAllAlerts: 100, failHighAlerts: 0, failMediumAlerts: 0, failLowAlerts: 0, falsePositivesFilePath: "zapFalsePositives.json")
        sh "docker rm -f vulnerable-application"

      }
    }
  }
}
```