Trabajo Fin de Máster
Ingeniería de Telecomunicación

Seguridad en la integración continua de la metodología ágil y la filosofía DevOps

Autor: Eleazar Rubio Sorrentino

Tutor: Juan Manuel Vozmediano Torres

Dep. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Trabajo Fin de Máster

Ingeniería de Telecomunicación

Seguridad en la integración continua de la metodología ágil y la filosofía DevOps

Autor:

Eleazar Rubio Sorrentino

Tutor:

Juan Manuel Vozmediano Torres

Profesor Titular

Dep. de Ingeniería Telemática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2017

Trabajo Fin de Máster: Seguridad en la integración continua de la metodología ágil y la filosofía DevOps

Autor: Eleazar Rubio Sorrentino
Tutor: Juan Manuel Vozmediano Torres

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

Apartado de agradecimientos.

Eleazar Rubio Sorrentino

Sevilla, 2017

Resumen

En los últimos tiempos de las empresas dedicadas al desarrollo de software como servicio (SaaS), los conceptos de metodología ágil y la filosofía Development and Operations (DevOps) están cobrando, cada vez más, un papel fundamental para el desarrollo de las mismas[15]. Según un estudio de alcance mundial realizado recientemente por CA Technologies, más del 75 por ciento de las organizaciones españolas coinciden en que las metodologías ágiles y DevOps son cruciales para el éxito de la transformación digital[26].

Este nuevo modo de entender el mundo del desarrollo de software (SW) posee una serie de elementos comunes, cada uno de ellos implementado con herramientas cada vez más conocidas y populares para las empresas que lo llevan a la práctica:

- Plataformas de desarrollo colaborativo y control de versiones de SW (por ejemplo GitHub[14]), donde se almacena el código desarrollado por las mismas.

- Diferentes entornos o infraestructuras de trabajo para los desarrolladores, que van a permitir un desarrollo y despliegue continuo para las mejoras del producto: entornos de desarrollo, entornos de seguro de calidad o Quality Assurance (QA), preproducción, producción o entorno final, etc.

- Fundamentos de Integración continua (IC) y Despliegue Continuo (DC).

- Desarrollo y de aplicaciones inmutables que utilizan contenedores de imágenes (generalmente de Docker[11]) para incrementar la protabilidad, reusabilidad y escalabilidad de la aplicación. Estas aplicaciones suelen mantener sus plataformas soportadas en Proveedores Cloud tales como Amazon Web Service (AWS)[24], Microsoft Azure[3] o Google Cloud Engine (GCE)[9].

El presente Trabajo Fin de Máster (TFM) pretende exponer un proceso automatizado de análisis estático de aplicaciones en varios niveles. El resultado de la ejecución periódica de estos procedimientos genera informes de seguridad que podrán ser utilizados para prevenir amenazas durante las fases más tempranas del Ciclo de Desarrollo Seguro (CDS) del software, advirtiendo de aspectos tales como:

1. Si la aplicación creada tiene Vulnerabilidades (Common Vulnerabilities and Exposures (CVE)[6]) en las librerías de dependencias de código utilizadas.

2. Si la imagen que se va a emplear para desplegar el contenedor de dicho SW contiene vulnerabilidades conocidas al nivel de Sistema Operativo (SO).

Abstract

Lately inside Software as a Service (SaaS) companies, the agile methodology and DevOps philosophy concepts are taking a fundamental role for the development of them[15]. According to a recent global survey by CA Technologies, more than 75 percent of Spanish organizations agree that DevOps and agile methodologies are crucial to the success of digital transformation[26].

This new way of understanding the world of SW development has various common elements, each of them implemented with well known and popular tools for the companies that put it into practice:

- Collaborative development platforms and SW version control systems (for example GitHub[14]), where the code developed is keeped.
- Different environments and infrastructures for developers, which will allow to continuous development and deployment for product enhancements: development environments, Quality Assurance QA environments, pre-production, production or final environment, etc.
- Continuous Integration (CI) and Continuous Deployment (CD) principles.
- Deployment and orchestration of immutable application images using containers (mostly Docker) that increases the portability, reusability and scalability of the application. These applications use Cloud Providers as the underlaying platform such as AWS[24], Microsoft Azure[3] or GCE[9].

This thesis tries to expose a procedure to automatically implement an application static analysis pipeline at several layers. The outcome of these periodically executed pipelines provide the company security reports that can be used to prevent threats during the earliest stages of the Secure Development Lifecycle (SDLC):

1. If the application created has Vulnerabilities (CVE[6]) in the code dependencies used.
2. If the image used to deploy the container of the mentioned SW contains known vulnerabilities at the Operative System (OS) level.

Índice

1 Introducción

You can ask 10 people for a definition of DevOps and likely get 10 different answers.

Dustin Whittle, 2014
Developer Advocate at Uber Developer Platform

En este primer apartado de la memoria se pretende realizar con el lector un recorrido a través del contexto que ha motivado el desarrollo del presente TFM para el Máster en Seguridad de la Información y las Comunicaciones de la Universidad de Sevilla, con el fin de aclarar el objetivo perseguido en su realización. Como conclusión al mismo, se definirá la estructura seguida durante la redacción, introduciendo cada uno de los apartados que se encontrarán a continuación.

1.1 Contexto y motivación

El año 2017, para empresas englobadas en todo tipo de sectores, está siendo el año de la transformación digital. Estos procesos de transformación exigen distintas formas de trabajo, más ágiles y colaborativas, con las que poder aplicar nuevas tecnologías que permitan conseguir los objetivos del negocio, en entornos que afrontan grandes desafíos culturales, organizativos y operativos e incluso pueden llegar a tener que lidiar con sistemas tecnológicos antiguos y casi obsoletos[2].

Es en este contexto donde el concepto DevOps empieza a sonar con más fuerza: el contexto de las metodologías ágiles.

De esta forma, DevOps es un concepto de trabajo, basada en el desarrollo de código, que usa nuevas herramientas y prácticas para reducir la tradicional distancia entre técnicos de programación y de sistemas, respondiendo a la necesidad experimentada por el sector tecnológico de dar una respuesta más rápida a la implementación y operación de aplicaciones. Este nuevo enfoque de colaboración que es DevOps permite a los equipos trabajar de forma más cercana, aportando mayor agilidad al negocio y notables incrementos de productividad.

Desde las pruebas de concepto hasta el lanzamiento, pasando por el *testing* y los entornos de prueba, todos los pasos involucrados requieren de la máxima agilidad posible (Figura 1.1), y eso pasa por integrar los procesos y los equipos de programación con los de sistemas[4].

Por otro lado, el concepto de contenedor de aplicación (aislamiento de espacio de nombres y gobernanza de recursos) a pesar de no ser un concepto novedoso, está cobrando cada vez más y más relevancia en el

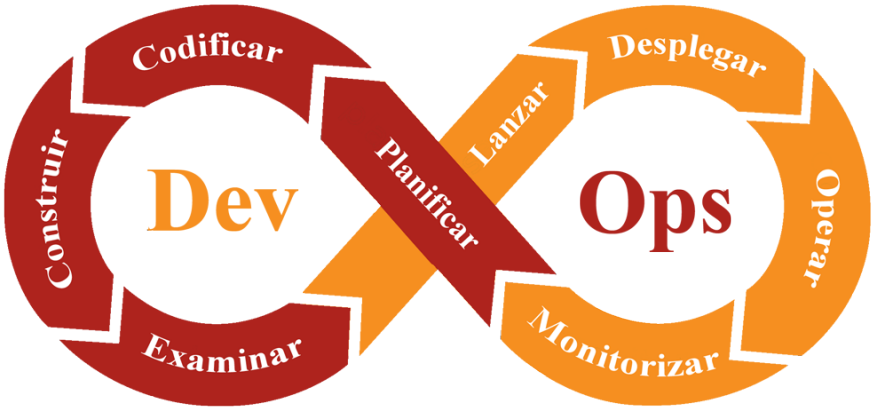

Figura 1.1 Introducción al proceso DevOps.

panorama de la empresa actual, de la mano de las continuas mejoras que experimentan las tecnologías que lo implementan, simplificando la administración y transformando la forma en que se desarrolla, distribuye y ejecuta el software, en forma de microservicio[1], además de proveer la habilidad de encapsular todo el entorno utilizado con el objetivo de ser desplegado en los sistemas de producción de la empresa, manteniendo las mismas características, aumentando la escalabilidad y disminuyendo notablemente los costes asociados a infraestructuras.

Los contenedores juegan un papel clave en un entorno DevOps porque soportan las implementaciones de la pila de desarrollo y operaciones completa y van en camino de formar parte de la definición básica de lo que se conocerá como DevOps en unos pocos años[8].

Además, la metodología DevOps representa una gran promesa a la hora de asegurar el desarrollo del software, ya que las organizaciones pueden potencialmente encontrar y remediar las vulnerabilidades con mayor frecuencia y al principio del ciclo de vida de la aplicación, ahorrando costes y tiempo. Conforme al informe de *"Seguridad de Aplicaciones y DevOps"* de octubre de 2016 promovido por Hewlett Packard Enterprise[10], que incluye tanto respuestas cualitativas como cuantitativas de profesionales de operaciones informáticas, líderes de seguridad y desarrolladores, se concluye que el 99% de los encuestados confirma que la adopción de la cultura DevOps aporta la oportunidad de mejorar la seguridad de las aplicaciones. Sin embargo, solo el 20% realizan análisis de seguridad de aplicaciones durante el desarrollo y el 17% no utilizan ninguna tecnología que proteja sus aplicaciones, destacando una desconexión significativa entre la percepción y la realidad de la seguridad DevOps.

Es en el contexto planteado donde surge la idea del presente TFM: aportar mecanismos a la metodología DevOps que permitan analizar la seguridad de las aplicaciones desarrolladas y el contenedor que las albergará dentro de la infraestructura de la empresa, sin interferir de manera destructiva con el propio proceso de desarrollo y depliegue de la aplicación.

1.2 Objetivo

El objetivo del presente Trabajo Fin de Máster (TFM) es desarrollar un entorno, basado en contenedores, que pueda ser incluido en el proceso de desarrollo e integración continua de la empresa y con el que poder realizar tareas periódicas programadas para analizar estáticamente las posibles vulnerabilidades de seguridad contenidas en las dependencias de aplicaciones desarrolladas mediante los lenguajes de programación Ruby y NodeJS, por ser dos de los lenguajes de programación más utilizados en el desarrollo de aplicaciones

[1] Aproximación para el desarrollo software que consiste en construir una aplicación como un conjunto de pequeños servicios, los cuales se ejecutan en su propio proceso y se comunican con mecanismos ligeros (normalmente una API de recursos HTTP)

Web, además de analizar a nivel del SO vulnerabilidades presentes en las imágenes que van a constituir el contenedor que dará soporte a dichas aplicaciones.

Como medio para alcanzar el objetivo planteado se va a hacer uso de una serie de aplicaciones y herramientas, entres las que cabe destacar las siguientes, que serán desarrolladas en los próximos apartados:

- GitHub[14]: Plataforma de desarrollo colaborativo y control de versiones de SW donde almacenar, entre otras, el código desarrollado y que será analizado.

- Jenkins CI[16]: Software de Integración Continua (IC) con el que automatizar los trabajos periódicos de análisis estático a realizar.

- Docker[11]: Proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de software, con el que desplegar los distintos elementos requeridos.

El trabajo aquí presentado no tiene como objetivo innovar en la tecnología existente, sino por contra valerse de esta para aportar al futuro usuario una herramienta intuitiva y de fácil aplicación, resultado de la agrupación de otras utilidades, con la que poder desplegar con el mínimo esfuerzo el entorno aquí recopilado.

1.3 Estructura de la memoria

Para facilitar la lectura de la memoria actual, se cree conveniente presentar un resumen de cómo se estructuran los diferentes apartados que contiene.

En el apartado actual, Introducción, se presenta el TFM que se va a realizar, aclarando el objetivo perseguido, el contexto en que surge y los aspectos que motivaron su realización.

El apartado 2, Descripción de la técnica, se pretende dar a conocer, de manera objetiva, las características de la realidad representada, con rasgos tales como elementos que la componen, utilidad, etc. En concreto, el apartado comienza describiendo en qué consiste la metodología Ágil, los mecanismo o procesos de Integración Continua IC y Despliegue Continuo DC, así como el Ciclo de Desarrollo Seguro de software (SDLC). Para concluir el apartado se aclarará al lector en qué consiste un análisis estático y qué es lo que se conoce como virtualización basada en contenedores.

El apartado 3, Entorno de trabajo, está dedicado a conocer las herramientas utilizadas para poder llevar a cabo el desarrollo y la implementación de la parte técnica de este TFM.

El apartado 4, titulado Desarrollo de la aplicación, es el apartado principal de la memoria. En él, se detalla el proceso a seguir durante el desarrollo e implementación de los objetivos presentados en este proyecto, comenzando con la preparación del entorno que se va a utilizar en el desarrollo, hasta llegar a presentar el Resultado final obtenido proceso de desarrollo del sistema.

Por último, el apartado 5, está dedicado a las Conclusiones y evaluaciones surgidas del proyecto, los objetivos alcanzados y las líneas futuras de trabajo surgidas durante la realización de éste.

2 Descripción de la Técnica

A good DevOps organization will free up developers to focus on doing what they do best: write software.

Rob Steward, 2015
Global Vicepresident at Verint-Systems

Para comprender el desarrollo del trabajo aquí presentado, tal y como se ha llevado a cabo, se debe conocer la situación en que éste se desarrolla, la tecnología de la que se dispone y los elementos existentes y necesarios, de una manera objetiva.

Es por esto, que el apartado actual está orientado a conocer las características de la realidad representada y a introducir las bases tecnológicas del presente TFM, resaltando los conceptos más importantes.

2.1 Metodología ágil

La tecnología actual avanza a una velocidad considerable, provocando a su paso la renovación de la gestión de proyectos informáticos, debiendo esta alcanzar la velocidad de los cambios ocasionados por esta aceleración.

Así, la calidad, eficiencia, rapidez y flexibilidad en la entrega de un determinado producto se ha convertido en prioritaria, dando paso a la conocida como metodología ágil.

La metodología ágil envuelve un enfoque para la toma de decisiones en los proyectos software que plantean métodos de ingeniería del software basado en el desarrollo iterativo e incremental, donde los requisitos y las soluciones evolucionan con el tiempo según la necesidad del proyecto. Así el trabajo es realizado mediante la colaboración de equipos auto-organizados y multidisciplinarios, inmersos en un proceso compartido de toma de decisiones a corto plazo[28].

El uso de procesos ágiles puede reportar los siguientes beneficios a la compañía:

- Flexibilidad en el proceso y las definiciones de los productos.
- Realimentación continua con el cliente.
- Iteracción constante del producto, que se va analizando a medida avanza.

- Calidad mejorada.

La Figura 2.1 muestra el ciclo de vida en cada iteracción que sufre la aplicación mediante la utilización de dicha metodología.

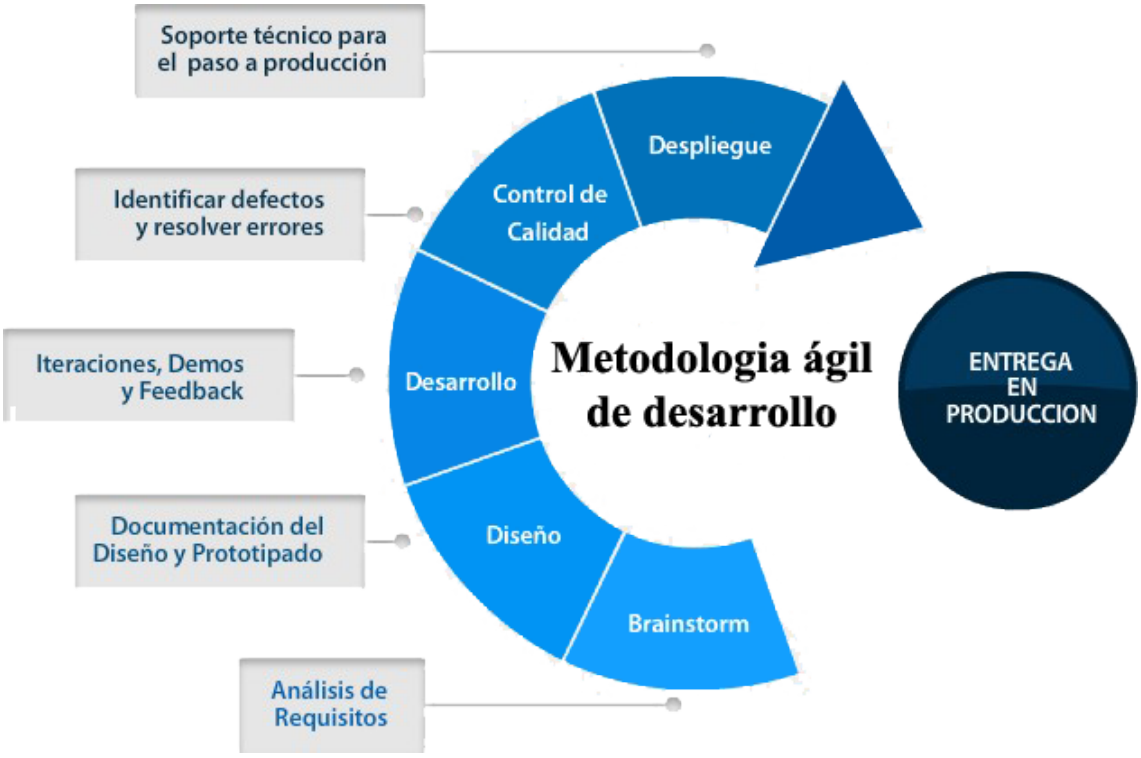

Figura 2.1 Ciclo de vida de la metodología ágil..

2.2 Integración Continua (IC) y Despliegue Continuo (DC)

Integración continua (IC) es una práctica de desarrollo que requiere que los desarrolladores integren nuevos cambios en el código de la aplicación varias veces en un sólo día. Cada vez que esto ocurre, la inserción es verificada por una compilación automática, permitiendo a los equipos de trabajo implicados en el proceso detectar cualquier problema que el código pudiera contener en cortos periodos de tiempo.

Integrando código regularmente los errores pueden ser detectados rápidamente y corregidos con más facilidad, debido a que se trata de un proceso muy frecuente, la búsqueda del error va a quedar muy acotada.

Por otro lado, el Despliegue Continuo (DC) está estrechamente relacionado a la IC y está referido a la liberación del código, en los entornos de producción de la compañía, que está sometido a pruebas continuamente y de forma automática.

Adoptar ambos conceptos (IC y DC) no solo reduce los riesgos y permite una localización temprana de fallos de código, sino que también permite aumentar la velocidad de trabajo con el SW[29].

2.3 Ciclo de Desarrollo Seguro de software (SDLC)

Un ciclo de Desarrollo Seguro de software (en inglés, Software Development Life Cycle, SDLC) es un marco de trabajo que define el proceso utilizado por las compañías a la hora de construir una aplicación desde sus inicios hasta el desmantelamiento de la misma. A lo largo de los últimos años,han surgido multitud de modelos para SDLC, que han sido utilizados de diversas maneras acorde a las circunstancias de cada aplicación o empresa en general. Las fases o etapas comunes a todos estos modelos para el Ciclo de Desarrollo de software son las siguientes:

- Requisitos y planificación.
- Arquitectura y diseño.
- Planificación de las pruebas.
- Desarrollo del código.
- Pruebas y resultados.
- Lanzamiento y mantenimiento.

Con esto, un proceso de implementación de seguridad en el proceso SDLC garantizará que las actividades de seguridad, como las pruebas de penetración, la revisión del código y el análisis de la arquitectura, son parte integral del esfuerzo de desarrollo.

2.4 Análisis estático

Análisis estático, o también conocido por análisis estático de código, es un método de depuración de aplicaciones que se basa en exámenes al código cuando éste no se está ejecutando. El proceso de análisis estático es realizado con la ayuda de herramientas automatizadas y asiste a los desarrolladores aportando valiosa información mediante el escrutinio del código, utilizando mecanismos y herramientas (por ejemplo revisión de impresiones y campos de formularios) que podrían escapar al ojo humano[1][20].

2.5 Dependencias de software

En el campo de la programación y el software una dependencia de código o SW es una aplicación o biblioteca de código requerida por otro programa para poder funcionar de manera correcta[25].

La lista de dependencias de SW que una aplicación ruby o una aplicación escrita con el lenguaje de programación node van a requerir se almacenan en archivos específicos llamados Gemfile.lock y package.json, respectivamente.

2.6 Virtualización basada en contenedores

La nube es cada vez más grande, más potente y posee más usuarios. Al mismo tiempo, permite la ejecución de aplicaciones más y más potentes, que requieren garantizar el correcto funcionamiento de esta, actualmente

[1] El análisis realizado por un humano recibe el nombre de comprensión de código

y en el futuro.

Por este motivo, es primordial utilizar una plataforma que optimice los recursos, en la medida de lo posible, a la vez que permita la escalabilidad, con el fin de poder ampliar sus características de forma sencilla en caso de ser necesario.

Al hablar de nube se está hablando de virtualización. Ejecutar un SO virtual para cada instancia de una aplicación es un proceso muy pesado y lento, es de este problema donde surge el concepto de virtualización basada en contenedores, también conocida como virtualización a nivel de sistema operativo.

Un contenedor no es más que una nueva forma de optimizar los recursos de los que dispone una plataforma, creando pequeños espacios virtuales de las aplicaciones necesarias, en las que unicamente se tendrá que cargar el núcleo de la aplicación y las dependencias de esta, pero funcionando siempre sobre un único kernel, o sitema operativo[27]. La Figura 2.2 muestra las diferencias en el modelo de capas de un sistema de virtualización tradicional y uno basado en contenedores.

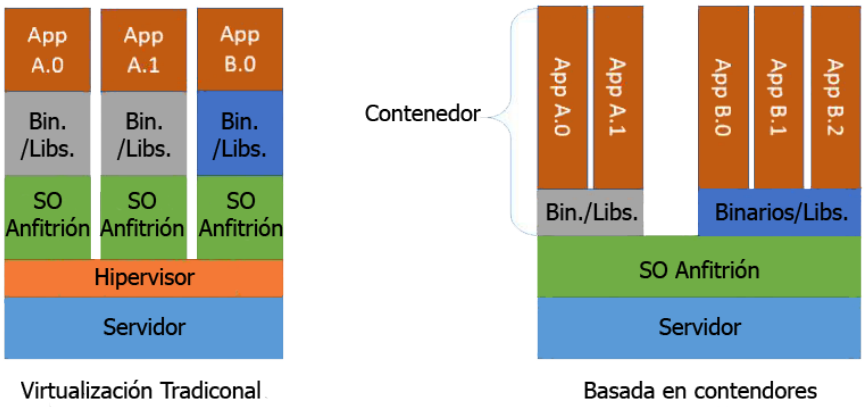

Figura 2.2 Sistemas de vitualización.

3 Entorno de trabajo

Technology is nothing. What's important is that you have a faith in people, that they're basically good and smart, and if you give them tools, they'll do wonderful things with them.

STEVE JOBS, 1994
BUSINESSMAN

Antes de comenzar el proceso de desarrollo de la parte técnica de la memoria es necesario preparar un entorno adecuado de trabajo, es decir, un conjunto de herramientas hardware y software que permitan llevar a cabo el proyecto con la mayor comodidad y precisión posible.

La correcta elección de un entorno de trabajo adecuado es fundamental a la hora de abordar cualquier tipo de proyecto, ya que el éxito o fracaso, o al menos la eficiencia del proceso de desarrollo del mismo, va a depender en gran medida de dicho entorno utilizado. El apartado actual presenta el entorno de trabajo utilizado para la realización de este TFM.

3.1 Git y GitHub

Los sistemas de control de versiones son programas cuyo principal objetivo es controlar los cambios producidos en el desarrollo de cualquier tipo de software, permitiendo conocer el estado actual de un proyecto, las personas que intervinieron en ellos, etc. Un buen control de versiones es tarea fundamental para la administración de un proyecto de desarrollo de software en general[1]. Git es uno de los sistemas de control de versiones más populares entre los desarrolladores, es gratuito, open source, rápido y eficiente, aunque gran parte su popularidad es debido a GitHub(Figura 3.7), un excelente servicio de alojamiento de repositorios de software que ofrece un amplio conjunto de características de gran utiilidad para el trabajo en equipo.

A continuación se muestran algunas de las características que han llevado a GitHub a ser tan valorado entre los desarrolladores[19]:

- Permite versionar el código, es decir, guardar en determinado momento los cambios realizados sobre un archivo o conjunto de archivos con la oportunidad de tener acceso al historial de cambios al completo, bien para regresar a alguna de las versiones anteriores o bien para poder realizar comparaciones entre ellas.

- Gracias a la gran cantidad de repositorios de SW públicos que alberga, es posible leer, estudiar

Figura 3.1 Logotipo de GitHub.

y aprender de el código creado por miles de desarrolladores en el mundo, permitiendo incluso la oportunidad de adaptarlos a las necesidades propias de cada desarrollador, sin alterar el original y realizando una copia o fork[1] de este.

- Tras haber realizado un fork de un proyecto y haber realizado algunos ajuste, introducido alguna mejora o arreglado algún problema que pudiera contener, es posible integrar los cambios realizados al proyecto original (previa supervisión de su propietario, administrador o alguno de sus colaboradores), por lo que un repositorio puede llegar a ser construido mediante la contribución de una gran comunidad de desarrolladores.

- GitHub posee un sistema propio de notificaciones con el que poder estar informado de lo que ocurre en torno a un repositorio concreto, ya sea privado a la compañía o público a la comunidad.

- GitHub trae incorporado un visor de código, mediante el cual (y a través del navegador) es posible consultar el contenido de un archivo determinado, con la sintaxis correspondiente al lenguaje utilizado y sin necesidad de descargar una copia del mismo.

- Cada repositorio de SW albergado en GitHub cuenta con su propio seguimiento de incidencias, con un elaborado sistema de tickets, de manera tal que cualquier colaborador (o usuario en general) pueda reportar algún problema encontrado en la utilización el código o pueda simplemente sugerir nuevas características para que sean implementadas.

- Al ser una plataforma web es totalmente independiente al SO utilizado, siendo por otro lado Git compatible con los principales sistemas actuales: Linux, Windows, OSX.

- GitHub es gratuito e ilimitado para repositorios de proyectos públicos, sólo aquellos usuarios que deseen mantener proyectos en privado deberán pagar una cuota.

3.2 Bundler-audit

Como ya fue comentado en el apartado 2.5 cada aplicación tiene sus dependencia, las que a su vez pueden contener vulnerabilidades de seguridad. Encontrar las vulnerabilidades de seguridad que presenta una aplicación es una tarea necesaria pero tediosa, que de ser obviada no impedirá que la aplicación generada siga siendo ejecutada como si todo estuviera funcionando en perfectas condiciones, pero ocultando agujeros de seguridad en la aplicación que podrán ser utilizados en diversa manera por algún usuario malintencionado.

Para cualquier aplicación escrita con Ruby y en ausencia de alguna herramienta automatizada de análisis de vulnerabilidades, el desarrollador del código deberá estar suscrito a cada lista de correo relacionada con anuncios de seguridad de dependencias y realizar un seguimiento exclusivo de vulnerabilidades y

[1] Copia exacta en crudo del repositorio original que podrá ser utilizada como un repositorio git cualquiera

actualizaciones de seguridad de cada una de las dependencias incluidas en la aplicación, para cada aplicación en la que participe[18].

Por el contrario, todo este proceso puede ser automatizado en Ruby gracias a bundler-audit[21], del grupo de colaboradores Rubysec, un verificador a nivel de parche para Bundler[2] con las siguientes características:

- bundler-audit analiza las vulnerabilidades en las versiones de las gemas contenidas en el archivo de dependencias *Gemfile.lock* de la aplicación.
- Analiza las fuentes de dependencias que puedan ser inseguras (http://).
- Permite especificar avisos de seguridad que serán ignorados en el análisis, bien por ser un riesgo asumido por el desarrollador, una vulnerabilidad ya conocida y en la que se está actualmente trabajando o cualquier otro motivo.
- No requiere de conexión a internet para cada ejecución que se realice del análisis.
- Funciona cruzando la información de dependencias recogidas del fichero *Gemfile.lock* con una lista de vulnerabilidades conocidas[22], basada en información pública existente en bases de datos como CVE[3].

El código 3.1 muestra un ejemplo del resultado obtenido a la salida del terminal de comandos al realizar un análisis estático de vulnerabilidades con bundler-audit al archivo *Gemfile.lock* de un proyecto:

Código 3.1 Ejemplo de uso de bundler-audit.

```
$ bundle audit
Name: actionpack
Version: 3.2.10
Advisory: OSVDB-91452
Criticality: Medium
URL: http://www.osvdb.org/show/osvdb/91452
Title: XSS vulnerability in sanitize_css in Action Pack
Solution: upgrade to ~> 2.3.18, ~> 3.1.12, >= 3.2.13

Name: actionpack
Version: 3.2.10
Advisory: OSVDB-89026
Criticality: High
URL: http://osvdb.org/show/osvdb/89026
Title: Ruby on Rails params_parser.rb Action Pack Type Casting Parameter
    Parsing Remote Code Execution
Solution: upgrade to ~> 2.3.15, ~> 3.0.19, ~> 3.1.10, >= 3.2.11

Name: activerecord
Version: 3.2.10
Advisory: OSVDB-90072
Criticality: Medium
URL: http://direct.osvdb.org/show/osvdb/90072
Title: Ruby on Rails Active Record attr_protected Method Bypass
Solution: upgrade to ~> 2.3.17, ~> 3.1.11, >= 3.2.12

Name: activerecord
```

[2] Bundler es la herramienta Ruby para el manejo de dependencias, mantenidas en los archivos Gemfile y Gemfile.lock.
[3] Lista de información registrada sobre vulnerabilidades conocidas, definida y mantenida por The MITRE Corporation

```
Version: 3.2.10
Advisory: OSVDB-89025
Criticality: High
URL: http://osvdb.org/show/osvdb/89025
Title: Ruby on Rails Active Record JSON Parameter Parsing Query Bypass
Solution: upgrade to ~> 2.3.16, ~> 3.0.19, ~> 3.1.10, >= 3.2.11

Unpatched versions found!
```

Además, bundler-audit permite actualizar la base de datos ruby-advisory-db y analizar el fichero *Gemfile.lock* desde el mismo comando, habilidad que resulta de gran utilidad para su ejecución en sistemas de IC, como muestra el código 3.2:

Código 3.2 Ejecutando bundler-audit tras la actualización de las vulnerabilidades conocidas.

```
$ bundle audit check --update
```

Por último, y como ya se ha mencionado con anterioridad en este apartado, es posible ignorar advertencias especificadas (código 3.3) por el usuario:

Código 3.3 Ignorar vulnerabilidades con bundler-audit.

```
$ bundle audit check --ignore OSVDB-108664
```

3.3 Nsp

Nsp es la principal herramienta de interfaz de línea de comandos de Node Security Platform[17] y es una de las herramientas más utilizadas para monitorizar la seguridad en aplicaciones node, permitiendo auditar el archivo *package.json* de dependencias de la misma.

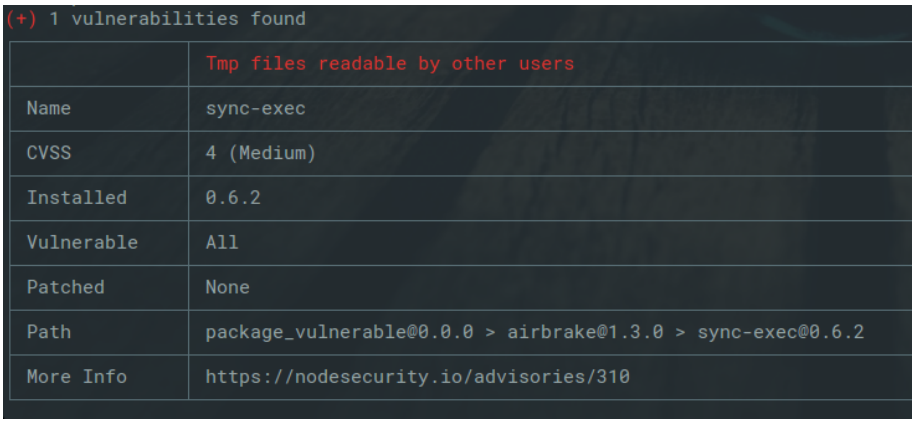

Figura 3.2 Escaneo de vulnerabilidades en las dependencias de Node JS.

Su utilización es simple, permitiendo incluso ignorar vulnerabilidades que puedan estar siendo tratadas en el momento del análisis y realizando consultas contra una base de datos de vulnerabilidades mantenida por Node Security Platform, que contiene información recopilada de múltiples fuentes. Basta con *"nsp*

check" para analizar las vulnerabilidades de la aplicación, recibiendo un resultado como el mostrado en la Figura 3.2.

3.4 Docker

Figura 3.3 Logotipo de Docker.

Docker (Figura 3.3) es una herramienta de virtualización diseñada para aportar beneficios a desarrolladores, testers y administradores de sistemas, creando contenedores ligeros y portables para que las aplicaciones puedan ser ejecutadas en cualquier máquina, con sólo tener docker instalado e independientemente del SO de la propia máquina que lo contenga[7]. Esta plataforma de código abierto hace uso de las funciones de aislamiento de recursos que provee el kernel de Linux para dar lugar a contenedores independientes, dentro de los cuales se ejecutará una única aplicación con sus respectivas dependencias, funcionando siempre con este único kernel, en lugar de virtualizar uno por cada contenedor o máquina virtual.

En sus orígenes, la tecnología Docker fue creada por encima de la tecnología LXC (término comúnmente relacionado con los contenedores Linux "tradicionales"), aunque actualmente está alejado de esta dependencia debido a que, a pesar de tener utilidad como virtualización ligera, no cuenta con un gran desarrollo como respaldo ni una gran experiencia de usuario. La tecnología Docker proporciona más que la capacidad de ejecutar contenedores, también facilita el proceso de creación y diseño de contenedores mediante el envío de imágenes y versionado de imágenes. La Figura 3.4 muestra una clara diferencias en las capas aportadas por cada uno de estos modos de virtualización de contenedores.

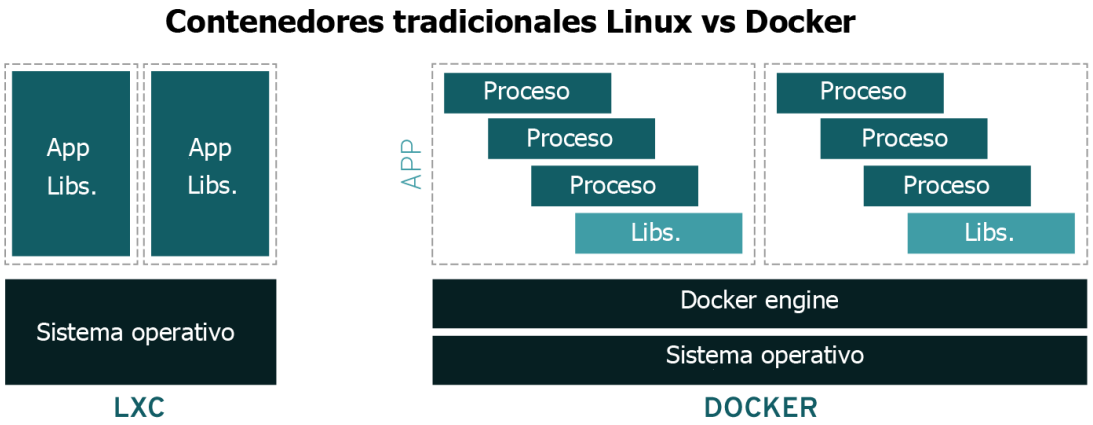

Figura 3.4 Contenedores Linux tradicionales vs Docker.

Gracias al uso de docker dos desarrolladores no tendrán que, por ejemplo, preocuparse de la versión Java que cada uno tenga instalado en su propia máquina durante el desarrollo de la aplicación, ya que está podrá ser enviada de uno a otro en el interior de un contenedor, que funcionará de la misma manera en cualquiera

de los dos entornos.

Además, virtualizar con docker ofrece una serie de ventajas respecto a hacerlo con máquinas virtuales convencionales[27]:

- **Portabilidad.** Los contenedores son portables, por lo que pueden ser trasladados fácilmente a cualquier otro equipo con Docker sin tener que volver a configurar nada.
- **Ligereza.** Al no virtualizar un sistema completo, sino solo lo necesario, el consumo de recursos es muy inferior.
- **Autosuficiencia.** Docker se encarga de todo de organizarlo todo, por lo que los contenedores tan solo deben tener lo necesario para que la aplicación funcione.

Un sistema de contenedores Docker lo componen principalmente 5 elementos fundamentales:

- **Demonio:** proceso principal de la plataforma.
- **Cliente:** binario que constituye la interfaz al usuario y le permite interactuar con el Demonio.
- **Imagen:** Plantilla utilizada para crear el contenedor de la aplicación que se va a ejecutar en su interior. Dicha plantilla recibe el nombre por defecto de *Dockerfile*.
- **Registros:** Directorios donde se almacenan las imágenes, tanto de acceso público como privado y similares a los utilizados por Linux, donde los usuarios publican sus propios contenedores de manera que aquellos otros usuarios que los necesiten puedan descargarlos directamente desde allí.
- **Contenedores:** Donde se almacena todo lo necesario (librerías, dependencias, binarios de la aplicación, etc.) para que la aplicación pueda ejecutarse de forma aislada.

Actualmente existen multitud de empresas que utilizan a docker como sistema de contenedores en sus centros de datos (Spotify, eBay, PayPal, etc.) y cuenta con el apoyo de grandes compañías de internet como Amazon o Google, lo que permite que Docker se encuentre en un proceso continuo de crecimiento y mejora.

3.5 Clair y Clairctl

Clair (Figura 3.5) es un proyecto de código libre desarrollado por CoreOS para el análisis estático de vulnerabilidades en contenedores de aplicaciones, que funciona de la siguiente manera[5]:

Figura 3.5 Clair.

1. A intervalos regulares, Clair realiza una ingesta de metadatos de vulnerabilidad desde la lista mantenida de vulnerabilidades CVE y los almacena en una base de datos PostgreSQL.

2. Un cliente configurado para utilizar la Interfaz de Programación de Aplicaciones (API) de Clair indexa y escanea sus imágenes de contenedores por capas, creando una lista de características presentes en la imagen y almacenándolas en la base de datos.

3. El cliente solicita a través de la API a la base de datos las vulnerabilidades conocidas para una imagen en particular, correlacionando las vulnerabilidades y características de la imagen, de manera que esta no requiere un escaneo completo para cada petición.

4. En caso de que se produzcan actualizaciones para los metadatos de vulnerabilidad, se enviará una notificación que alertará a los sistemas de que se ha producido un cambio, siendo escaneada de nuevo la imagen la próxima vez que sea solicitado un informe.

Por otro lado, Clairctl es uno de los clientes más utilizados para alcanzar la API de Clair y se caracteriza por ser una herramienta ligera de interfaz de comandos y escrita en el lenguaje de programación desarrollado por Google, Go. Clairctl es capaz de hacer de puente entre registros de imágenes tales como Docker Hub, Docker Registry o Quay.io (registro de imágenes de CoreOS). La Figura 3.6 muestra el esquema resultado de utilizar ambas herramientas para el análisis estático de contenedores de aplicaciones.

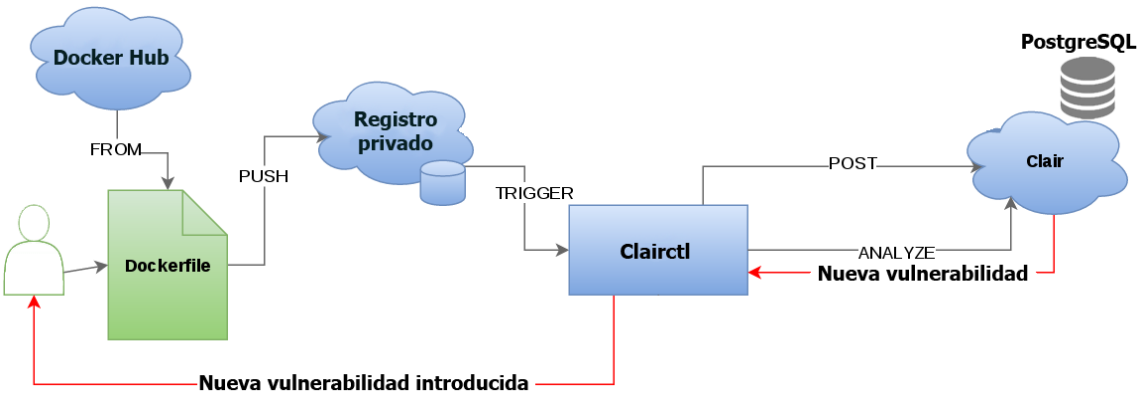

Figura 3.6 Esquema de escaneo y detección de vulnerabilidades en imágenes Docker.

3.6 Jenkins CI

Jenkins CI (Figura 3.7) es una herramienta autónoma de código abierto que puede utilizarse para automatizar todo tipo de tareas, como la construcción, prueba y despliegue de software. Jenkins puede ser instalado a través de paquetes de sistemas nativos, Docker, o incluso ser ejecutado de manera independiente en cualquier máquina con el entorno de ejecución de Java instalado[16].

Jenkins posee, entre otras, las siguientes ventajas:

- **Integración Continua y Despliegue Continuo**: Al ser un servidor de automatización extensible, Jenkins puede ser utilizado simplemente como un servidor de IC o convertirse en un entorno completo de IC y DC para cualquier tipo de proyecto.

- **Fácil instalación**: Jenkins es una aplicación auto-contenida diseñada con Java, lista para ser ejecutada y empaquetada para los SO Windows y Mac OSX, así como cualquier sistema Unix.

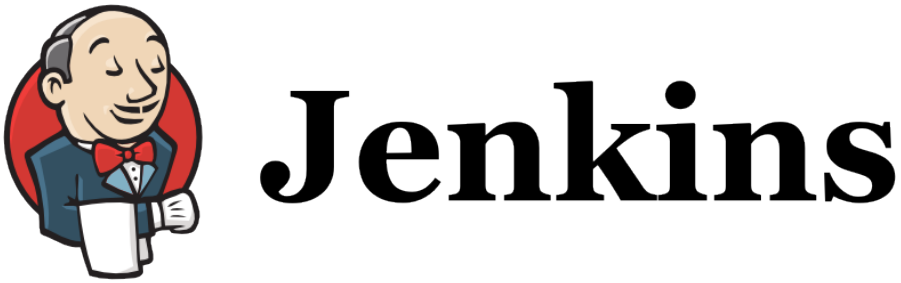

Figura 3.7 IC con Jenkins.

- **Fácil configuración**: Jenkins aporta una configuración mediante interfaz web de fácil manejo, que incluye análisis instantáneos de errores y ayuda en compilaciones.
- **Extensible**: Jenkins dispone de cientos de plugins en su centro de actualizaciones, capaz de ser integrados prácticamente con cualquier herramienta necesaria en los procesos de integración continua y despliegue continuo.
- **Distribuido**: Jenkins puede distribuir fácilmente el trabajo en varias máquinas, lo que ayuda a generar mejoras, pruebas y despliegues en varias plataformas rápidamente.
- **Open Source**: Es una herramienta de código libre con el apoyo de una gran comunidad.
- **Soporte de repositorio de control de versiones**: Gracias a Jenkins, el código puede ser descargado directamente desde un repositorio de contro de versiones (GitHub, BitBucket, etc.) con multitud de opciones.
- **Soporte de scripts**: Puede ejecutar scripts de bash, shell, ANT y proyectos Maven.
- **Notificaciones**: Jenkins puede ser utilizado para publicar resultados y enviar emails de notificación de los mismos.

La Figura 3.8 muestra el aspecto del panel de ejecuciones de líneas de trabajo o pipelines, con múltiples tareas, en Jenkins.

Figura 3.8 Pipeline de trabajo en Jenkins.

3.7 Slack

Slack (Figura 3.9) es una herramienta de mensajería en tiempo real diseñada para usarla en grupo, facilitando la gestión de los propios grupos de trabajo dentro del entorno de una empresa y centralizando la comunicación interna a la misma[23]. Además, permite la creación de grupos de trabajo con un número ilimitado de miembros, a la vez que es posible establecer canales de comunicación públicos, conversaciones privadas y compartir ficheros.

Figura 3.9 Slack.

Por otro lado, Slack permite a sus usuarios trabajar con otros servicios basados en la web, amplificando aún más el trabajo colaborativo y proveyendo integración con múltiples herramientas de uso habitual: Dropbox, Trello, Google Drive, GitHub y Jenkins entre otras.

4 Desarrollo de la solución

As an engineer, you learn there is a solution to every problem.
It may take you a while, but eventually you're going to find it.

Tony Cárdenas
American politician

El apartado actual está dedicado a comprender el proceso seguido durante el desarrollo de la solución realizada. Comenzando con una introducción que sitúe al lector en condiciones que le permitan reproducir el proceso presentado y los primeros pasos a seguir en la preparación del entorno, desplegando los sistemas que componen dicha solución, para así continuar automatizando el proceso en vistas a un uso prolongado en el tiempo. Por último, se evaluará la solución obtenida en un proceso de verificación de la solución mostrando el resultado final obtenido.

4.1 Preparación del entorno de trabajo

4.1.1 Docker

Como ya ha quedado reflejado en apartados anteriores, las nuevas habilidades proporcionadas por la virtualización mediante contenedores de aplicación permiten desplegar el sistema compuesto de múltiples contenedores con independencia al SO en el que estos se ejecuten, siendo el único requisito indispensable contar con el motor de virtualización de docker instalado y ejecutándose en la máquina. Por esto, toda solución desarrollada a partir de virtualización de contenedores comenzará con la instalación de docker en su sistema.

Por otro lado, se va a hacer uso de Docker compose, una herramienta de los creadores de Docker diseñada para permitir la definición, ejecución y seguimiento de aplicaciones Docker compuesta por múltiples contenedores. Será con la ayuda de Docker Compose como se despliegue el sistema utilizado.

No se cree necesario detallar en esta memoria el proceso seguido en la instalación de dichos elementos, ya que la propia documentación aportada para Docker[12] y Docker Compose[13] en los canales oficiales incluye una guía de como llevarla a cabo para cada SO, aunque es necesario comprender que cualquier reproducción del entorno desplegado comenzará siguiendo los pasos ofrecidos en dicha documentación.

4.1.2 Docker Compose

Esta sección tiene por objetivo especificar los elementos relevantes que conforman al archivo *docker-compose.yml*, necesario para poder automatizar la tarea de despliegue de todos los elementos. Para visualizar el contenido completo de este archivo, así como los archivos de configuración de los servicios que acompañan a este, se remite al lector al apéndice A (Repositorio de GitHub), donde encontrará el acceso a la ubicación de todos los ficheros utilizados para el desarrollo del presente TFM.

En primer lugar, el Código 4.1 muestra el fragmento de información que Docker Compose va a necesitar para ejecutar un contenedor (se va a usar Clair como ejemplo, por ser el más completo) que albergue a una aplicación. Cada elemento representa lo siguiente:

Código 4.1 Definiendo Clair en Docker Compose.

```
clair:
  container_name: clair_clair
  image: quay.io/coreos/clair:v2.0.0
  restart: unless-stopped
  depends_on:
    - postgres
  ports:
    - "6060-6061:6060-6061"
  links:
    - postgres
  volumes:
    - /tmp:/tmp
    - ./clair_config:/config
  command: [-config, /config/config.yaml]
```

- **container_name:** nombre que recibirá el contenedor una vez se encuentre en ejecución.
- **image:** ubicación de la imagen de Docker que va a ser desplegada. Este parámetro puede contener bien una referencia a un repositorio de imágenes Docker o bien a la ubicación del archivo *Dockerfile*, en la propia máquina, con las instrucciones para de creación de dicha imagen.
- **restart:** instrucciones de reinicio del contenedor ante imprevistos.
- **depends_on:** con la instrucción *depends_on* Docker Compose entenderá que el contenedor que se está desplegando depende de que exista otro (en el ejemplo actual postgres) en ejecución y no intentará ejecutar un contenedor sin la presencia del otro.
- **ports:** traducción de puertos ”red_externa:red_docker_virtualizada” necesaria para poder acceder al servicio desplegado desde el exterior del sistema Docker.
- **links:** permite conectar el contenedor con el servicio que se ejecuta en otro contenedor de forma que será accesible desde el hostname o el alias referenciado (http://postgres en el ejemplo actual).
- **volumes:** especifica la ruta o el volumen de Docker que será montado en el nuevo contenedor y la ruta destino a montar, en la forma ”ruta_origen:ruta_destino”.
- **command:** comando que ejecutará el contenedor una vez arrancado.

Visto lo anterior, el fragmento de Código 4.2 contiene las instrucciones necesarias para desplegar la base de datos PostgreSQL que necesitará Clair para su funcionamiento donde, por no ser alcance de este

proyecto, la contraseña que requerirá PostgreSQL para su funcionamiento está incluida como variable de entorno en el propio contenedor, suceso que bajo ningún concepto debe ocurrir en un sistema en producción. La documentación oficial de cada uno de los sistemas aquí tratados incluye información suficiente para reemplazar los métodos de conexión en claro por certificados SSL e intercambios de Tokens que permitan una conexión segura en un entorno distribuido.

Código 4.2 Definiendo PostgreSQL en Docker Compose.

```
postgres:
  container_name: clair_postgres
  image: postgres:9.6
  restart: unless-stopped
  volumes:
    - "data-postgres:/var/lib/postgresql/data"
  environment:
    POSTGRES_PASSWORD: password
```

Para el contenedor que contendrá Jenkins CI (Código 4.3) se necesitará además montar como volumen los binarios de Docker y el socket del demonio Docker, para proveer a Jenkins de la habilidad de ejecutar docker y ejecutar comandos del cliente Docker contra el resto de los contenedores desplegados:

Código 4.3 Definiendo Jenkins en Docker Compose.

```
jenkins:
  build: ./jenkins
  container_name: jenkins
  ports:
    - "80:8080"
  restart: "always"
  volumes:
    - "data-jenkins:/var/jenkins_home"
    - "reports:/reports"
    - "/usr/bin/docker:/usr/bin/docker"
    - "/var/run/docker.sock:/var/run/docker.sock:ro"
    - "/usr/lib/x86_64-linux-gnu/libltdl.so.7:/usr/lib/x86_64-linux-gnu/libltdl
        .so.7"
```

Por último, las herramientas de análisis estáticos podrían no ser definidas en el archivo *docker-compose.yml*, puesto que no son servicios de los que Docker Compose vaya a requerir mantener control constante, sino más bien aplicaciones clientes que serán ejecutadas en ciertos instantes de tiempo. Aún así, se ha decidido que incluir la definición de dichos contenedores entre la configuración del resto de los servicios facilita al usuario la tarea de despliegue inicial de las herramientas, debido a que será la propia herramienta Docker Compose la encargada de preparar en la máquina anfitriona las imágenes y volúmenes necesarios para la ejecución. El Código 4.4 muestra lo comentado.

Código 4.4 Definiendo Las herramientas de análisis estático en Docker Compose.

```
clairctl:
  build: ./clairctl
  container_name: clair_clairctl
  command: [echo, Hello from clairctl container]

ruby-tool:
  build: ./audit-tools/ruby
  container_name: ruby-tool
```

```
  command: [echo, Hello from ruby-tool container]

nodejs-tool:
  build: ./audit-tools/nodejs
  container_name: nodejs-tool
  command: [echo, Hello from nodejs-tool container]
```

Una vez definidos todos los elementos, la ejecución del sistema se resume en la instrucción ”docker-compose up -d” desde la carpeta contenedora del fichero *docker-compose.yml* en la máquina. Este comando mantendrá en ejecución y en segundo plano (-d) los servicios mencionados, creando las imágenes de Docker que pudiera necesitar para ello. La Figura 4.1 muestra la información aportada por el sistema en funcionamiento.

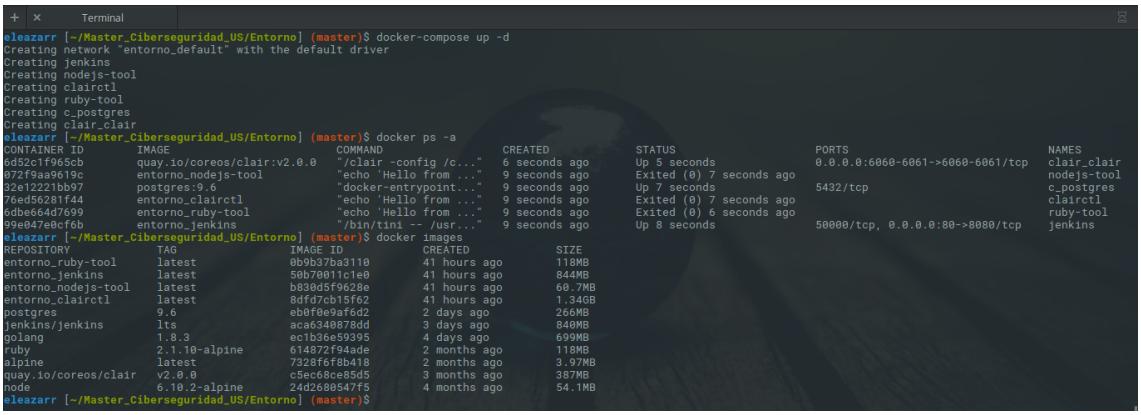

Figura 4.1 Depliegue del entorno con Docker Compose.

4.1.3 Probando los componentes desplegados

Ahora que todos los sistemas están funcionando en el sistema de contenedores es momento de comprobar que la configuración proporcionada es la correcta y que el comportamiento es el esperado. Para ello, y antes de automatizar las tareas con la ayuda de Jenkins, se van a realizar desde el interfaz de comandos las instrucciones necesarias que confirmen que los análisis se están llevando a cabo sin problema por parte de las herramientas de análisis.

El Código 4.9 muestra la ayuda aportada por el comando ”docker exec”, utilidad proporcionada por Docker para la ejecución de instrucciones en el interior de los contenedores que se encuentran en ejecución en el entorno (Jenkins, clair_clair y c_postgres).

Código 4.5 Comando de ayuda de docker exec.

```
$ docker exec --help

Usage: docker exec [OPTIONS] CONTAINER COMMAND [ARG...]

Run a command in a running container

Options:
  -d, --detach         Detached mode: run command in the background
  --detach-keys string  Override the key sequence for detaching a container
  -e, --env list       Set environment variables
  --help               Print usage
```

```
  -i, --interactive    Keep STDIN open even if not attached
  --privileged         Give extended privileges to the command
  -t, --tty            Allocate a pseudo-TTY
  -u, --user string    Username or UID (format: <name|uid>[:<group|gid>])
```

Para las imágenes de contenedores que no estén en ejecución Docker incluye la utilidad ”docker run”, con multitud de opciones y argumentos que se adaptan a las ejecuciones más variadas que puedan requerir las imágenes del sistema. El Código 4.9 ejemplifica el uso de la instrucción que será necesario llevar a cabo en el entorno actual presentado.

Código 4.6 Comando de ayuda de docker exec.

```
$ docker run --help

Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

...

$ sudo docker run --rm -t -i -v ruta_local:ruta_contenedor imagen:tag /bin/bash
root@394bb66c4df0:/home/contenedor# pwd
/home/contenedor
```

Donde:

- *--rm* indica al demonio de Docker que una vez se termine de trabajar en el contenedor creado, este debe ser eliminado del sistema.
- *-t -i* sirve para especificar que se va a abrir un terminal (-t) interactivo (-i) con el contenedor.
- *-v* irá continuado por el volumen que se va a montar en el interior del contenedor de Docker.
- *imagen:tag* será la imagen utilizada para crear el contenedor.
- */bin/bash* va a ser la instrucción ejecutada una vez creado el contenedor. Este comando va a devolver al usuario un terminal bash interactivo desde el interior del contenedor, donde poder ejecutar las instrucciones necesarias.

Conocidos estos comando y utilidades será posible comprobar la funcionalidad del sistema montado.

Bundler Audit

El Código 4.11 muestra los pasos a seguir para comprobar que Bundler Audit funciona correctamente desde el contenedor de Docker.

Código 4.7 Probando bundle-audit desde el contenedor.

```
eleazarr [~/Master_Ciberseguridad_US/Entorno] (master)$ docker run --rm -ti
    entorno_ruby-tool:latest sh

/ # curl https://raw.githubusercontent.com/EleazarWorkshare/gemfile_vulnerable/
    master/Gemfile.lock -o ./Gemfile.lock -s
/ # curl https://raw.githubusercontent.com/EleazarWorkshare/gemfile_vulnerable/
    master/Gemfile -o ./Gemfile -s
```

```
/ # bundle-audit check
Name: activesupport
Version: 4.1.1
Advisory: CVE-2015-3226
Criticality: Unknown
URL: https://groups.google.com/forum/#!topic/ruby-security-ann/7VlB_pck3hU
Title: XSS Vulnerability in ActiveSupport::JSON.encode
Solution: upgrade to >= 4.2.2, ~> 4.1.11

Name: activesupport
Version: 4.1.1
Advisory: CVE-2015-3227
Criticality: Unknown
URL: https://groups.google.com/forum/#!topic/rubyonrails-security/bahr2JLnxvk
Title: Possible Denial of Service attack in Active Support
Solution: upgrade to >= 4.2.2, ~> 4.1.11, ~> 3.2.22

Vulnerabilities found!
```

Analizando la ejecución en detalle:

1. Se accede al contenedor que incluye la herramienta Bundle Audit.

2. Mediante la API de GitHub se descarga desde el repositorio que se quiere analizar los ficheros *Gemfile* y *Gemfile.lock* que incluyen las dependencias de la aplicación examinada.

3. ”bundle-audit check” examina los ficheros, descubriendo que existen vulnerabilidades en la versión 4.1.1 de la dependencia *activesupport* contenida en el repositorio.

NSP

El funcionamiento de nsp en el terminal de comando es muy similar al de bundler-audit, como demuestra la Figura 4.2.

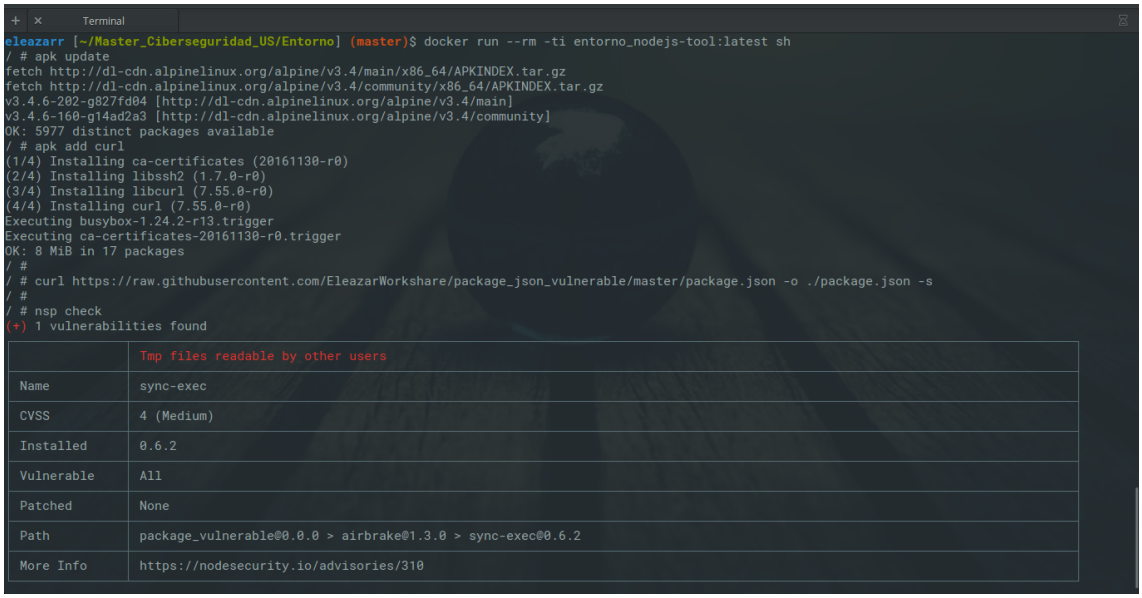

Figura 4.2 Comprobando la utilidad nsp.

La Figura 4.2 sigue el proceso de descargar un archivo de dependencias package.json desde un repositorio de GitHub y analizarlo para descubrir que, efectivamente, este contiene una vulnerabilidad.

Con lo que queda demostrado que es posible desde el contenedor que ejecuta nsp acceder a la API de GitHub, descargar un archivo de dependencias de la aplicación node y realizar un análisis sobre este.

Clair y Clairctl

La forma de funcionamiento de Clairctl con Clair es diferente a las vistas para Bundler Audit y NSP, ya que Clairctl requiere de una configuración apropiada que le permita comunicarse con Clair (Apéndice A) y deberá realizar una serie de pasos para ello:

1. Enviará la imagen que se quiere analizar a Clair para que la almacene (push). Dicha imagen podrá encontrarse en la máquina local que contiene a Clairctl o en un repositorio de imágenes de internet (pull).

2. Clairctl pedirá a Clair que analiza la imagen enviada y le devuelva el resultado de dicho análisis (analyze).

3. Clairctl generará un informe en formato HTML que podrá ser visualizado en el navegador del usuario.

El proceso comentado se traduce en un mínimo de dos instrucciones de terminal para Clairctl: una para enviar y analizar la imagen en Clair y otra para realizar los informes. El Código 4.8 muestra el resultado de estas ejecuciones, tomando como ejemplo la propia imagen que contiene a Clairctl.

Código 4.8 Generando un informe HTML con Clairctl.

```
eleazarr [~/Master_Ciberseguridad_US/Entorno] (master)$ docker run --rm -ti -v
    /home/eleazarr/informe_clairctl:/go/src/github.com/jgsqware/clairctl/
    reports -v /var/run/docker.sock:/var/run/docker.sock:ro --net
    entorno_default --link clair_clair:clair -p 57330:57330 entorno_clairctl
    bash
root@59d0b5391fc6:/go# cd src/github.com/jgsqware/clairctl/
root@59d0b5391fc6:/go/src/github.com/jgsqware/clairctl# clairctl health

Clair: OK

root@59d0b5391fc6:/go/src/github.com/jgsqware/clairctl# ./clairctl analyze --
    local entorno_clairctl:latest
2017-09-03 18:03:45.717118 I | config: retrieving interface for local IP
2017-09-03 18:03:45.717535 I | server: Starting Server on 172.20.0.4:57330
2017-09-03 18:03:45.722495 I | config: retrieving interface for local IP
2017-09-03 18:03:45.722730 I | clair: using http://172.20.0.4:57330/local as
    local url
2017-09-03 18:03:45.722748 I | clair: Pushing Layer 1/9 [b3ddc1248fc3]
2017-09-03 18:03:46.651866 I | clair: Pushing Layer 2/9 [b7f7dd16a445]
2017-09-03 18:03:46.654461 I | clair: Pushing Layer 3/9 [85ac2ce41a64]
2017-09-03 18:03:46.656823 I | clair: Pushing Layer 4/9 [089137f68fc4]
2017-09-03 18:03:46.658893 I | clair: Pushing Layer 5/9 [382f1b17bd70]
2017-09-03 18:03:46.660915 I | clair: Pushing Layer 6/9 [fef71688ca46]
2017-09-03 18:03:46.662980 I | clair: Pushing Layer 7/9 [71e3be3b4746]
2017-09-03 18:03:46.664839 I | clair: Pushing Layer 8/9 [76ceb8887997]
2017-09-03 18:03:46.666559 I | clair: Pushing Layer 9/9 [5ee4b01f03bd]
2017-09-03 18:03:46.668400 I | config: retrieving interface for local IP
```

```
2017-09-03 18:03:46.668611 I | clair: using http://172.20.0.4:57330/local as
    local url
2017-09-03 18:04:03.116083 I | clair: analysing layer [5ee4b01f03bd] 1/9
2017-09-03 18:04:03.143454 I | clair: analysing layer [76ceb8887997] 2/9
2017-09-03 18:04:03.190460 I | clair: analysing layer [71e3be3b4746] 3/9
2017-09-03 18:04:03.212756 I | clair: analysing layer [fef71688ca46] 4/9
2017-09-03 18:04:03.238765 I | clair: analysing layer [382f1b17bd70] 5/9
2017-09-03 18:04:03.285257 I | clair: analysing layer [089137f68fc4] 6/9
2017-09-03 18:04:03.302773 I | clair: analysing layer [85ac2ce41a64] 7/9
2017-09-03 18:04:03.316155 I | clair: analysing layer [b7f7dd16a445] 8/9
2017-09-03 18:04:03.325171 I | clair: analysing layer [b3ddc1248fc3] 9/9

Image: docker.io/entorno_clairctl:latest

Unknown: 15
Negligible: 65
Low: 49
Medium: 83
High: 37
Critical: 0
Defcon1: 0
root@59d0b5391fc6:/go/src/github.com/jgsqware/clairctl# ./clairctl report --
    local entorno_clairctl:latest

HTML report at ./reports/html/analysis-entorno_clairctl-latest.html
```

La Figura 4.3 muestra el resultado generado por Clairctl en el análisis de su propia imagen de Docker, confirmando que el sistema al completo se encuentra en perfecto estado de funcionamiento.

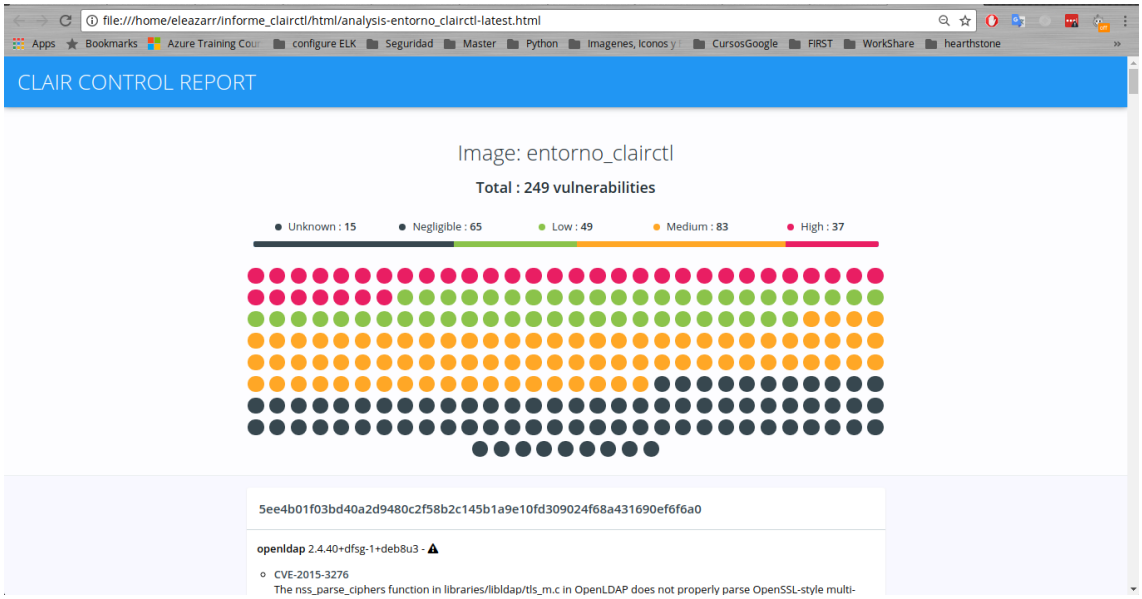

Figura 4.3 Inorme generado por Clairctl.

Jenkins CI

Para concluir el apartado se va a confirmar que existe el acceso apropiado al panel de trabajo de Jenkins, así como que se dispone de los plugins necesarios instalados para poder automatizar los trabajos de Jenkins, más conocidos por su traducción al inglés Jenkins Jobs.

En primer lugar, la Figura 4.4 muestra el mensaje de bienvenida que el usuario encontrará la primera vez que accede a la URL de la aplicación, solicitando un código oculto en el sistema de archivos de la misma. La Figura 4.5 muestra el acceso al código necesario para comenzar a crear el primer usuario de Jenkins.

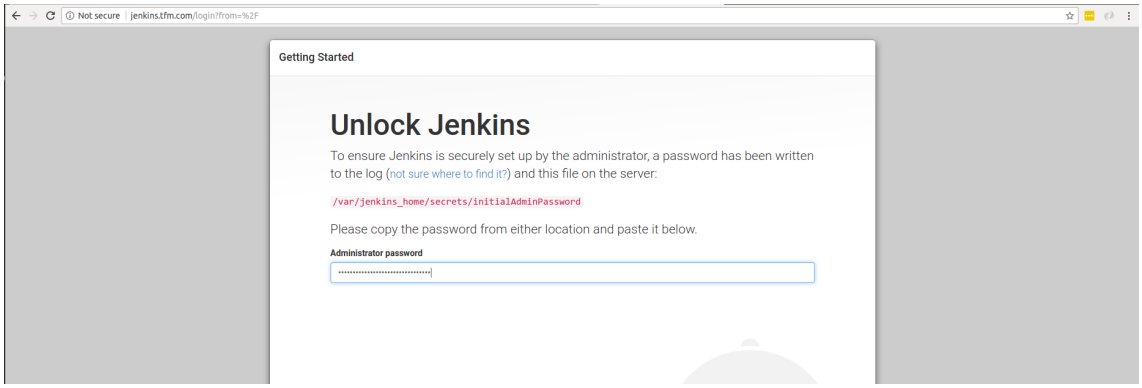

Figura 4.4 Bienvenida a la aplicación.

```
eleazarr [~] $ docker images
REPOSITORY                  TAG                 IMAGE ID            CREATED             SIZE
entorno_ruby-tool           latest              e696be922b90        About a minute ago  118MB
entorno_jenkins             latest              dceb43b3976d        About a minute ago  844MB
entorno_nodejs-tool         latest              0d972eb29a61        2 minutes ago       60.7MB
entorno_clairctl            latest              7dce6682cbff        3 minutes ago       1.37GB
quay.io/coreos/clair-git    latest              02c150a7eb41        14 hours ago        422MB
postgres                    latest              eb0f0e9af6d2        14 hours ago        266MB
jenkins/jenkins             lts                 aca6340878dd        38 hours ago        840MB
golang                      latest              5e2f23f821ca        2 days ago          727MB
ruby                        2.1.10-alpine       614872f94ade        2 months ago        118MB
node                        6.10.2-alpine       24d2680547f5        4 months ago        54.1MB
eleazarr [~] $ docker exec  -ti jenkins bash
jenkins@4f58433cdacf:/$ cat /var/jenkins_home/secrets/initialAdminPassword
4c23c9dad3c74e52b869ef223e635c8c
jenkins@4f58433cdacf:/$
```

Figura 4.5 Encontrando el código oculto.

Una vez superado la pantalla de bienvenida a la aplicación, se podrá decidir los plugins que serán instalados al inicio (Figura 4.6) y comenzará el proceso de instalación inicial de plugins (Figura 4.7).

El último paso de la preparación inicial será crear al primero de los usuarios administrador de la aplicación, la Figura 4.8 muestra al usuario creado.

Con esto la aplicación queda configurada y puede comenzar a usarse (Figura 4.9).

Una vez Jenkins está configurado, lo primero que se deberá hacer para poder llevar a cabo el objetivo del presente TFM será instalar el plugin *HTML_publisher*, puesto que será utilizando este como se crearán los informes HTML resultado de la ejecución de cada análisis estático. Las figuras 4.10 y 4.11 muestran las pantallas recorridas.

Tras aplicar la instalación del plugin, y como muestra la Figura 4.12, Jenkins se reiniciará, devolviendo al usuario al panel principal (Figura 4.13) de la aplicación, con el entorno completamente configurado para comenzar a añadir los trabajos de Jenkins que automatizarán el proceso de análisis.

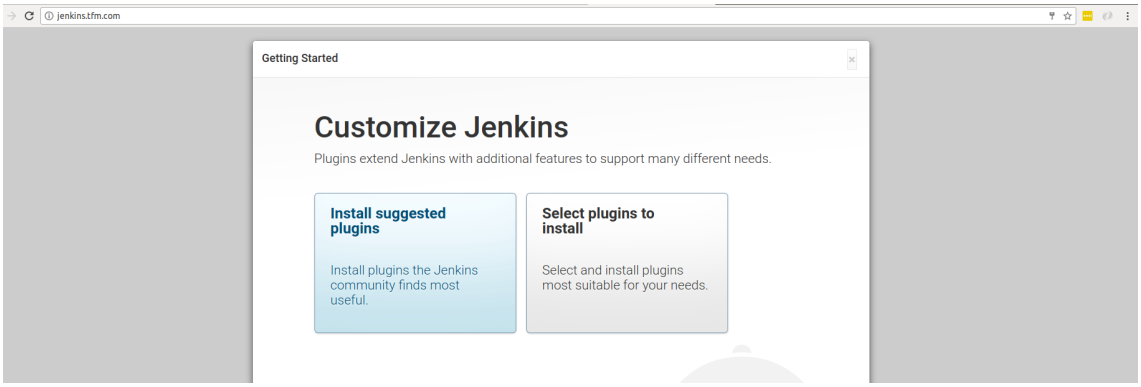

Figura 4.6 Instalando plugins recomendados.

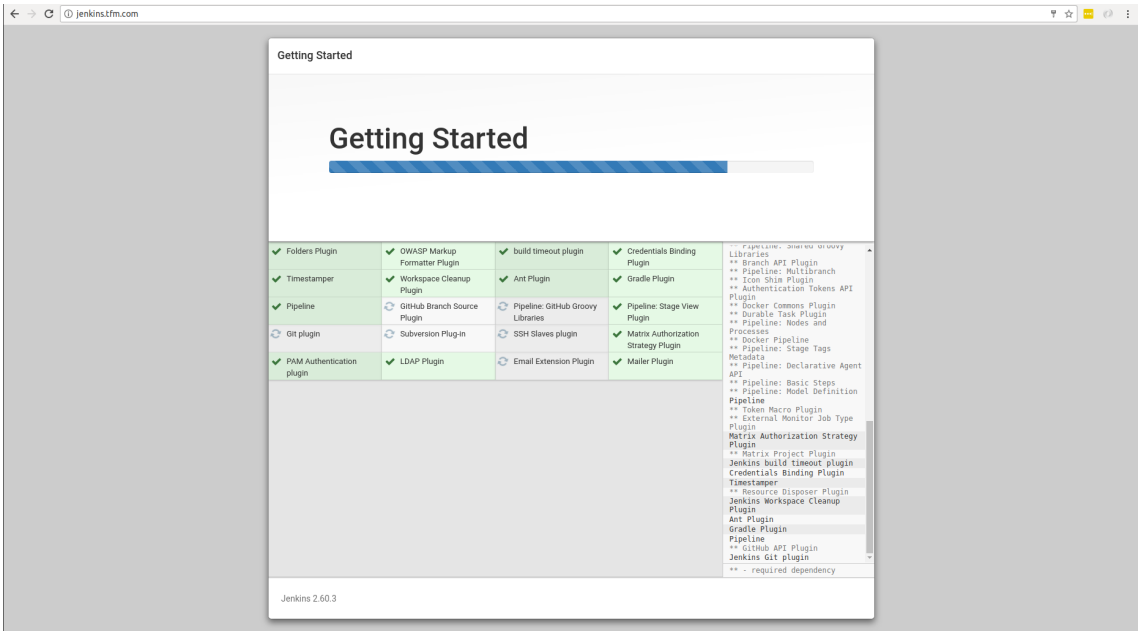

Figura 4.7 Proceso de instalación de plugins.

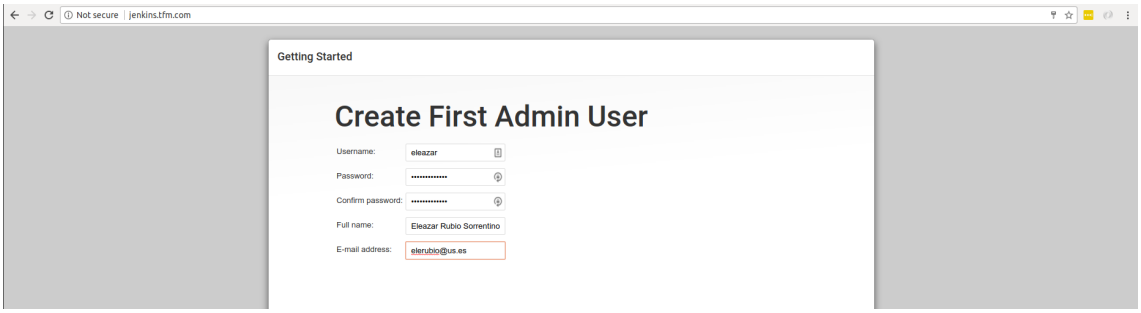

Figura 4.8 Creando el primer usuario administrador.

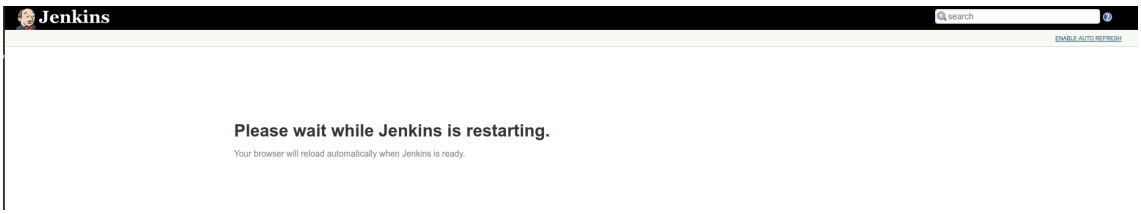

Figura 4.12 Reiniciando la aplicación tras la instalación del plugin.

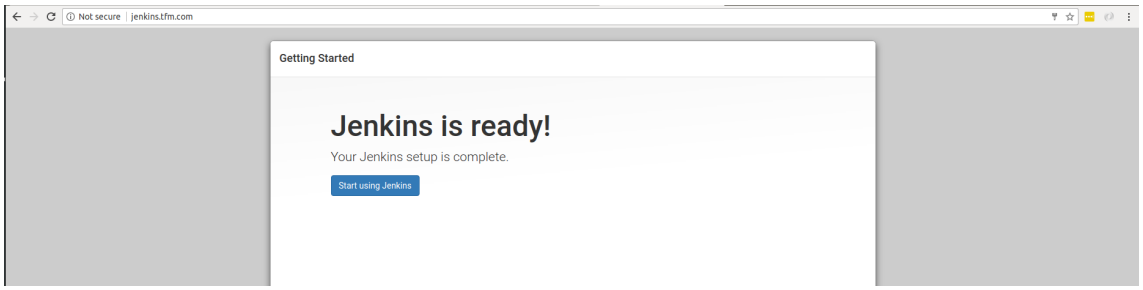

Figura 4.9 !Jenkins está listo!.

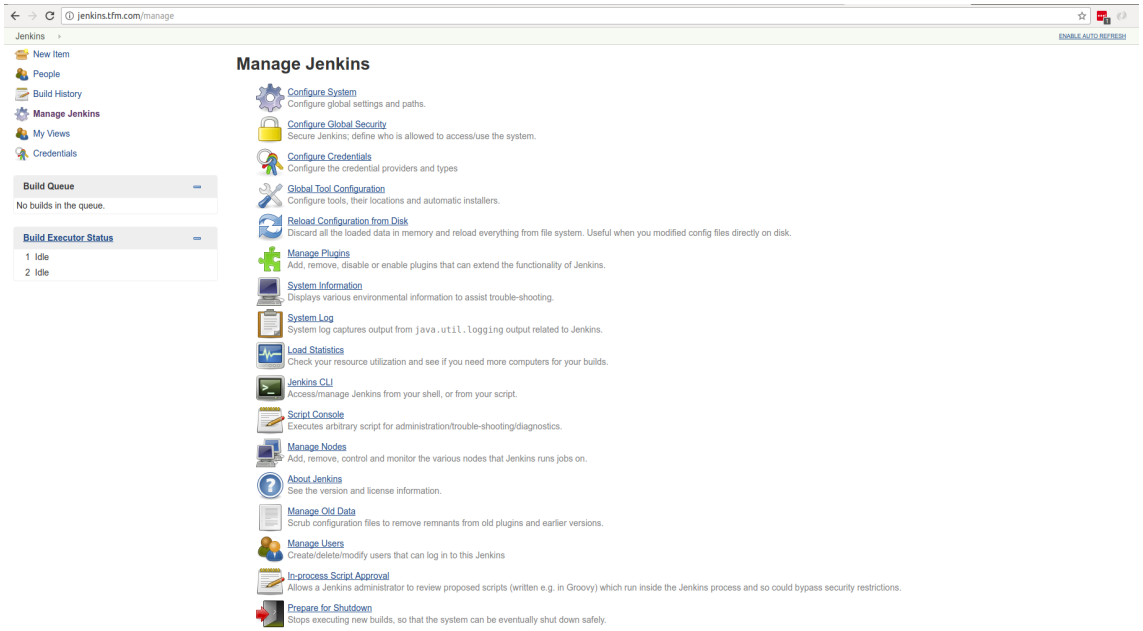

Figura 4.10 Manage Jenkins -> Manage Plugins.

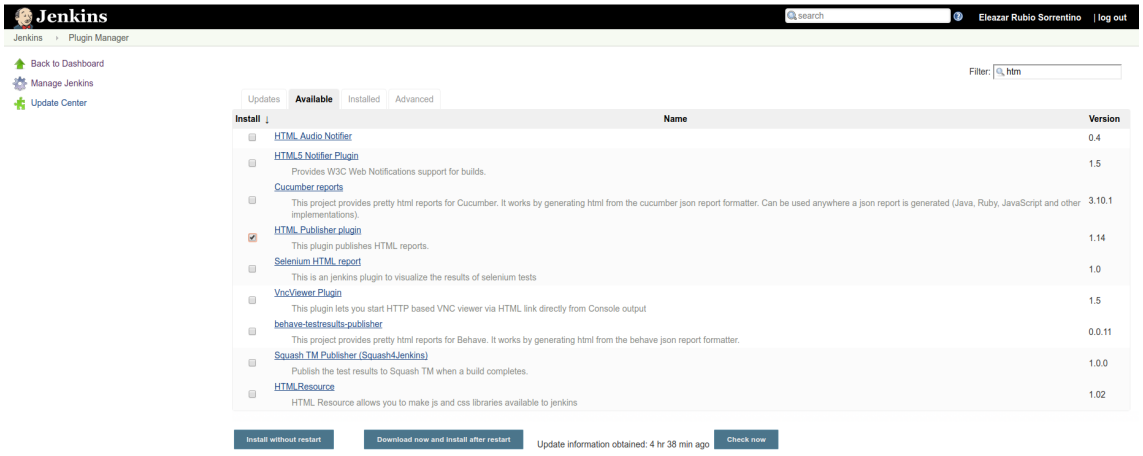

Figura 4.11 Instalando HTML_publisher.

4.2 Trabajando con Jenkins

En la sección actual se va a utilizar el contenido empleado en los apartados de la Sección 4.1 para mostrar al lector la creación de tareas o trabajos en la plataforma de IC Jenkins, con el fin de automatizar el proceso

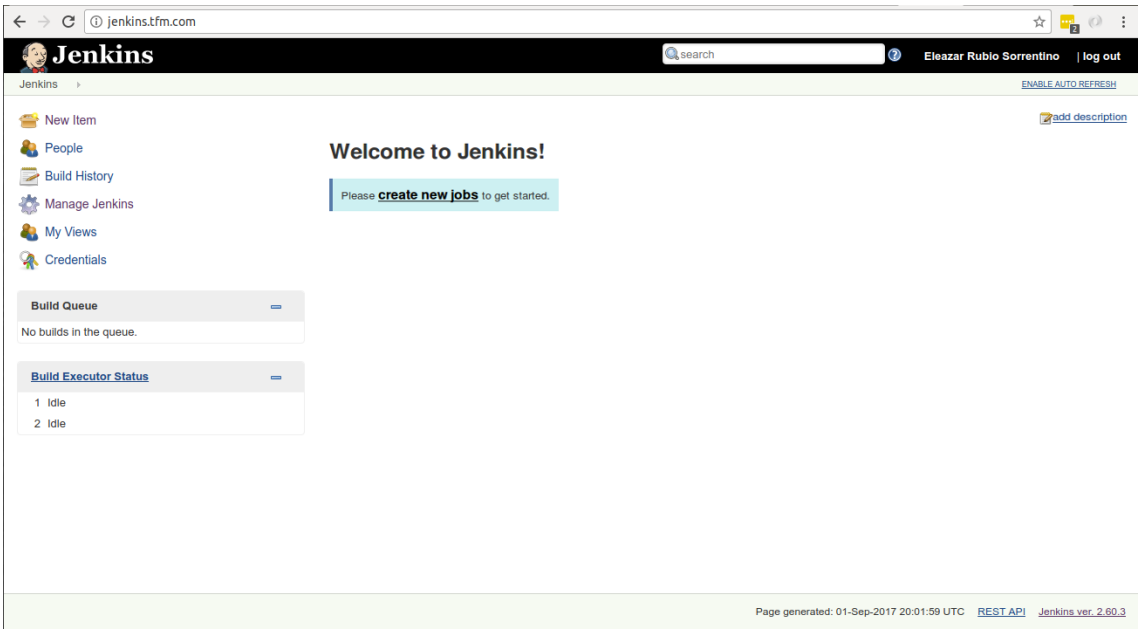

Figura 4.13 Pantalla de inicio de Jenkins CI.

de análisis de las dependencias de código para aplicaciones escritas mediante los lenguajes de programación Ruby y Node.js, asi como el análisis de las posibles vulnerabilidades que pudieran mantener los contenedores de Docker utilizados por la compañía.

4.2.1 Análisis de dependencias Ruby y Node.js

Como se pudo comprobar en la sección 4.1.3, el modo de utilización de las herramientas y componentes necesarios para la comprobación de las dependencias de código de ruby y node y los pasos ha seguir durante el proceso son similares, por lo que el apartado actual va a describir de forma exhaustiva la creación del trabajo de Jenkins para el análisis de dependencias de código en ruby, para terminar mostrando el resultado de la ejecución de ambos trabajos (ruby y node) en Jenkins.

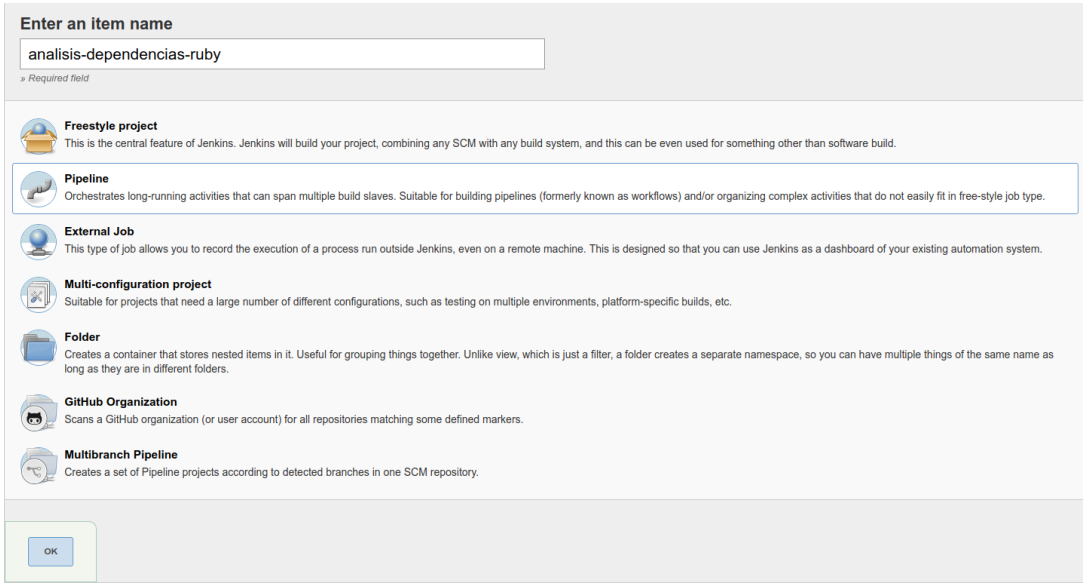

Figura 4.14 Creando el nuevo trabajo de Jenkins ”analisis-dependencias-ruby”.

En primer lugar es necesario crear el nuevo trabajo de Jenkins, para eso desde la pantalla principal de la aplicación se debe hacer clic en la opción ”New Item”. La Figura 4.14 muestra la pantalla de creación de nuevo trabajo, en el proceso de crear un trabajo tipo pipeline llamado ”analisis-dependencias-ruby”.

A continuación, Se definirán aspectos tales como los parámetros de entrada, la dependencia con repositorios externos o la periodicidad con la que la tarea será ejecutada automáticamente. La Figura 4.15 muestra lo comentado, permitiendo definir para la entrada una variable que contendrá los repositorios a analizar, así como una ejecución automática diaria para la misma.

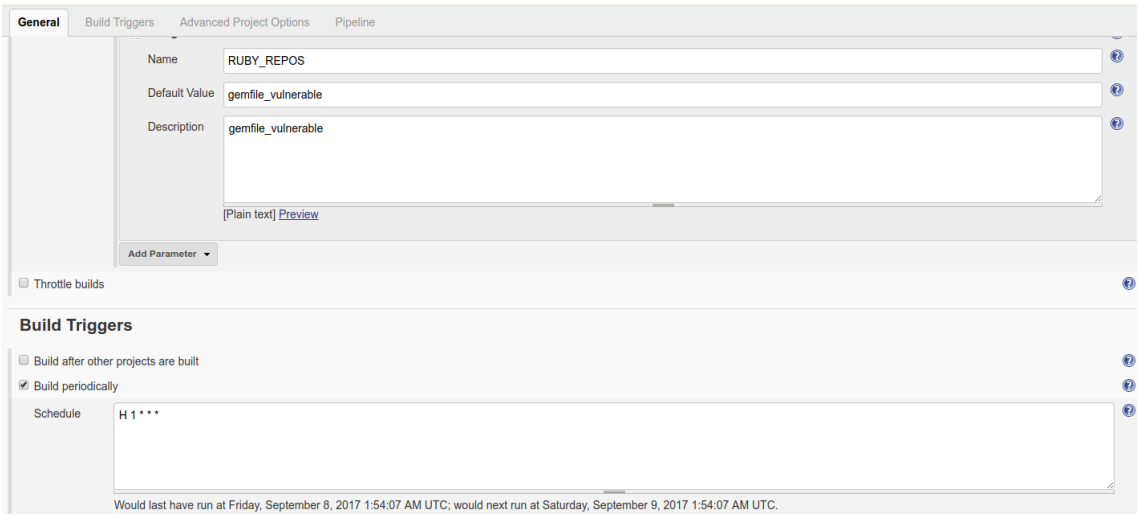

Figura 4.15 Configurando la tarea.

Por último, se describe la funcionalidad que la tarea va a realizar, utilizando el lenguaje de programación Groovy, un lenguaje de programación orientado a objetos implementado sobre la plataforma Java. El código utilizado para programar la tarea puede ser encontrado mediante la información proporcionada en el Apéndice A, por lo que no será presentado en su totalidad en esta memoria. Las fases de ejecución seguidas para definirla son los siguientes:

1. Utilizando la API proporcionada por GitHub y la herramienta de interfaz de comandos, se van a descargar los ficheros Gemfile y Gemfile.lock del repositorio indicado en el parámetro de entrada, así como el archivo *.bundle-auditrc* utilizado para seleccionar aquellas vulnerabilidades que se van a obviar. El Código 4.9 muestra la instrucción *curl* construida para ello.

Código 4.9 Curl al repositorio GitHub.

```
curl https://raw.githubusercontent.com/EleazarWorkshare/${RUBY_REPOS[i]}/
    master/${audit_files[j]} -o ./${RUBY_REPOS[i]}/${audit_files[j]} -f
```

2. Se ejecuta la instrucción del Código 4.10 para ejecutar el contenedor de docker que contiene la herramienta Bundler Audit, cuyo resultado de la ejecución será devuelto a Jenkins para procesar el contenido y poder convertir el resultado a HTML.

Código 4.10 Ejecutando el análisis de dependencias.

```
sudo docker run --rm -i -v /var/lib/docker/volumes/entorno_data-jenkins/
    _data/workspace/analisis-dependencias-ruby/${RUBY_REPOS[i]}/Gemfile.
    lock:/Gemfile.lock -v /var/lib/docker/volumes/entorno_data-jenkins/
    _data/workspace/analisis-dependencias-ruby/${RUBY_REPOS[i]}/Gemfile:/
```

```
        Gemfile entorno_ruby-tool:latest bundle-audit check --update ${ignore
        }
```

3. Desde la tarea de Jenkins se crea y almacena el informe HTML que podrá analizar posteriormente el usuario.

4. Se notifica vía Slack de la existencia del nuevo análisis, aportando el enlace directo al informe HTML generado.

5. En caso de encontrar vulnerabilidades en las dependencias examinadas, el trabajo realizado por Jenkins será marcado como trabajo fallido, lo que dibujará la ejecución de la tarea de color rojo, indicando al usuario que el resultado no es el deseado.

Figura 4.16 Panel principal del trabajo analisis-dependencias-ruby.

Una vez almacenada la tarea y desde el panel principal de la misma, la opción ”Build with Parameters” permitirá su ejecución. La Figura 4.16 muestra el aspecto de dicho panel tras varias ejecuciones.

Y el informe generado será el de la Figura 4.17

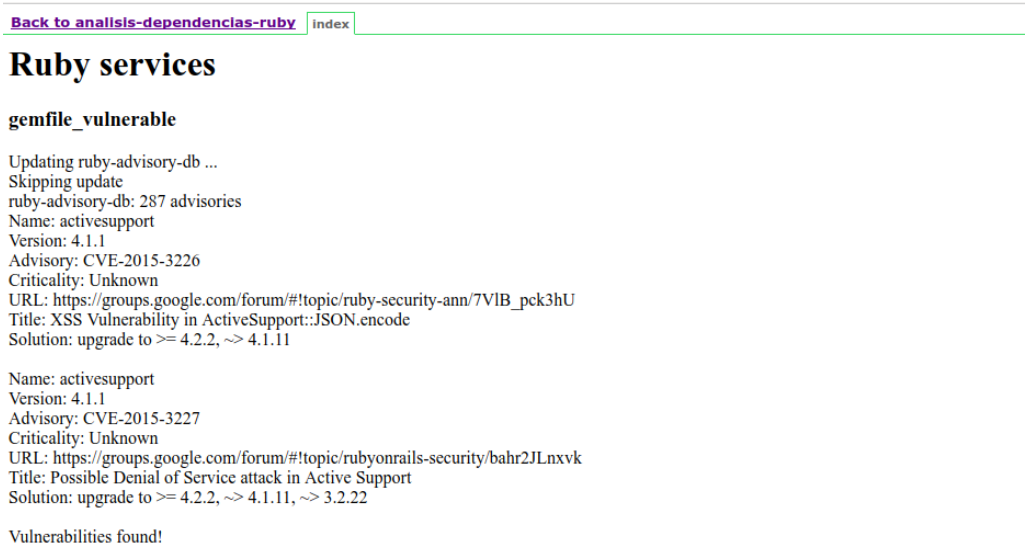

Figura 4.17 Informe de vulnerabilidades en dependencias ruby.

De tratarse del análisis de dependencias para Node.js, el volumen montado en la instrucción del Código 4.10 será ”package.json”, así como el nombre de la imagen utilizada ”entorno_nodejs-tool” con la opción de ejecución ”nsp check −−output json −−warn-only”. Las figuras 4.18 y 4.19 muestran el panel principal de la tarea ”analisis-dependencias-nodejs” tras varias ejecuciones, así como el informe HTML aportado por la misma.

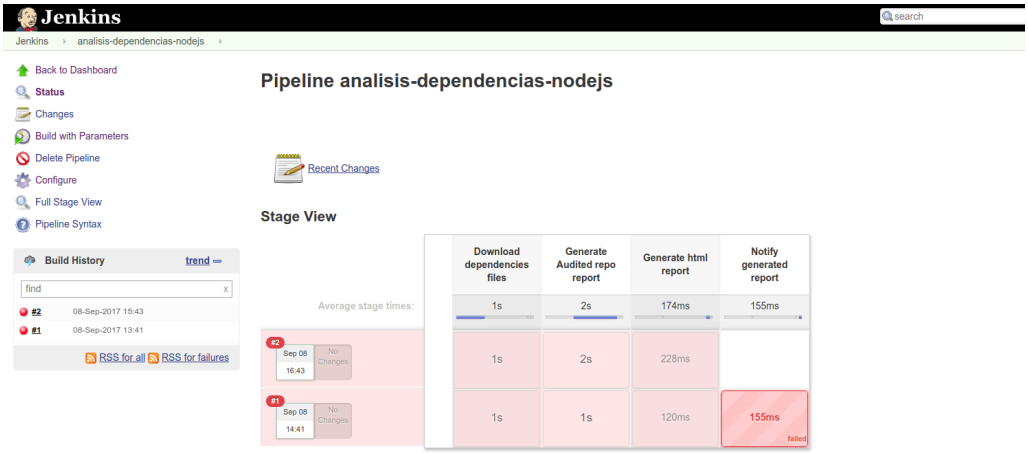

Figura 4.18 Panel principal del trabajo analisis-dependencias-nodejs.

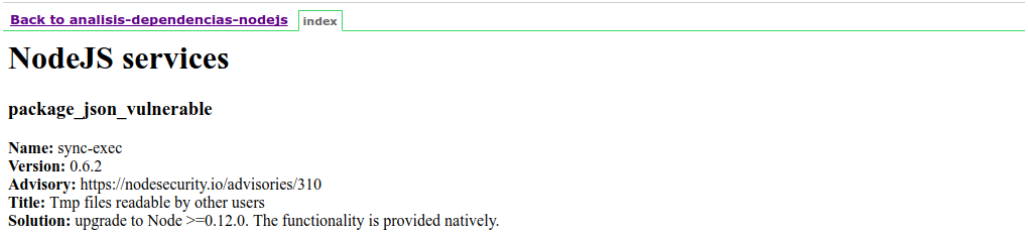

Figura 4.19 Informe de vulnerabilidades en dependencias node.

4.2.2 Análisis de imágenes Docker

La siguiente tarea que se va a configurar es la encargada de realizar los análisis estáticos sobre las imágenes de Docker, ”analisis-imagenes-docker”.

Para ello de vuelta a la pantalla principal de la aplicación Jenkins y mediante la opción ”New Item”, se va a crear una nueva tarea de tipo ”pipeline” como la comentada en el apartado anterior. La Figura 4.20 muestra los parámetros de entrada que la tarea va a admitir por defecto, además de la configuración del periodo en que dicho análisis será ejecutado automáticamente (una vez al día).

Para el trabajo actual se han seleccionado por defecto el análisis de aquellas imágenes que componen el sistema desplegado, lo que permitirá identificar las vulnerabilidades que contiene esta herramienta.

Las fases que componen el trabajo actual de Jenkins son:

1. Como muestra el Código 4.11, se invoca al contenedor que alberga a la herramienta Clairctl con los parámetros necesarios para enviar, analizar y devolver el informe resultado del análisis, así como con

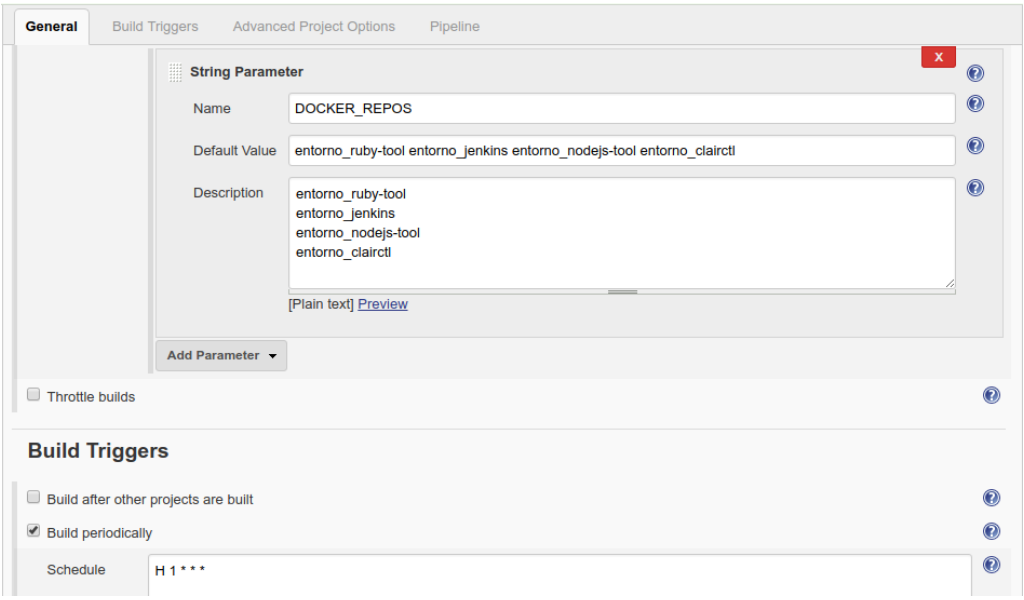

Figura 4.20 Configurando la tarea de análisis estático de imágenes Docker.

la configuración de volúmenes, sockets e interfaces de red que van a permitir la correcta comunicación entre Clairctl y Clair. Además de habilitar una ubicación donde Clairctl podrá alojar los informes de manera tal que Jenkins pueda recuperar el contenido y procesarlo para obtener la vista HTML de los mismos.

Código 4.11 Ejecutando Clairctl desde el trabajo de Jenkins.

```
sudo docker run --rm -t -v entorno_reports:/go/src/github.com/jgsqware/
    clairctl/reports -v /var/run/docker.sock:/var/run/docker.sock:ro --
    net entorno_default --link clair_clair:clair -p 57330:57330
    entorno_clairctl bash -c "cd /go/src/github.com/jgsqware/clairctl/ ;
    ./clairctl analyze --log-level info --local $repos ; ./clairctl
    report --log-level info --local $repos"
```

2. Se almacenan los informes obtenidos como resultado de la ejecución anterior bajo el plugin *HTML Publisher* de jenkins.

3. Por último, el resultado del análisis es notificado al canal de Slack correspondiente.

Las figuras 4.21 y 4.22 muestran la apariencia del panel principal del trabajo de Jenkins y el resultado de visualizar uno de los informes generados, respectivamente.

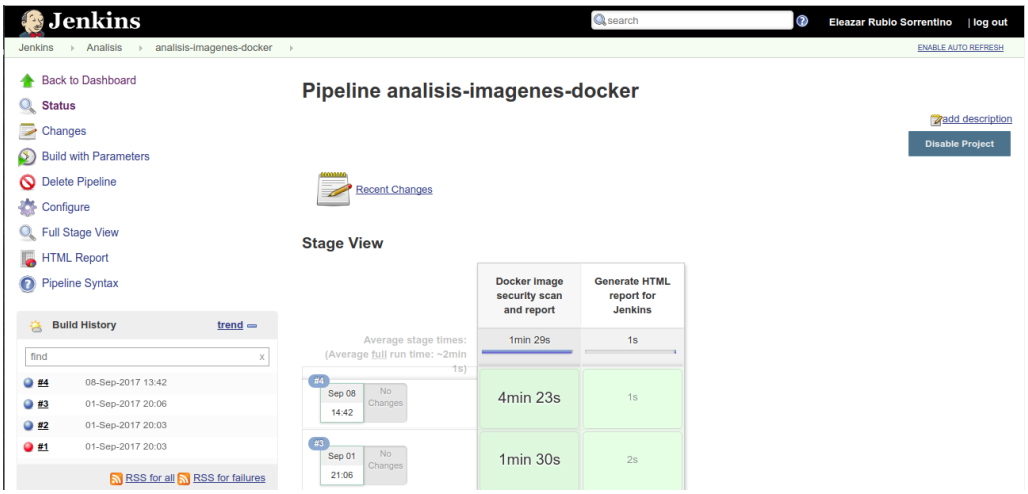

Figura 4.21 Ejecuciones del trabajo ”analisis-imagenes-docker”.

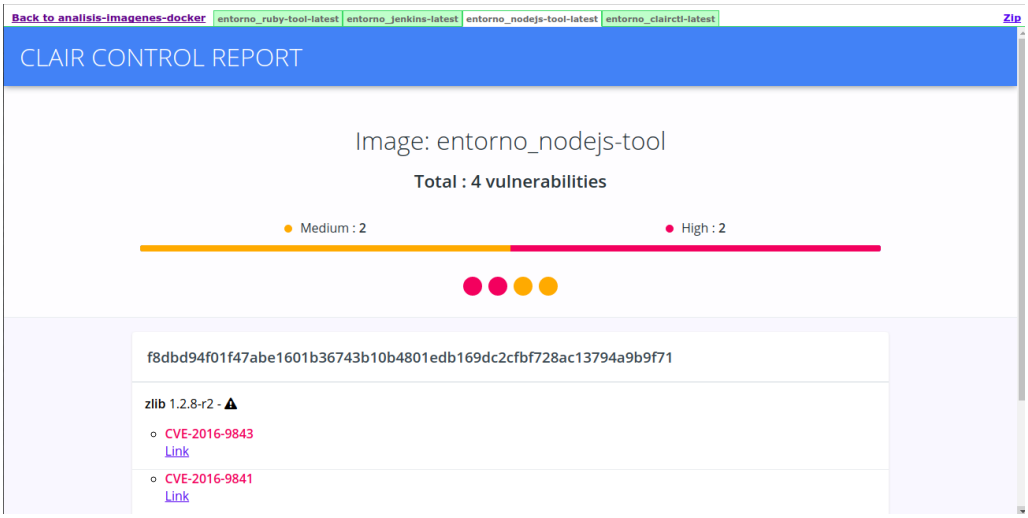

Figura 4.22 Resultado del informe de vulnerabilidades.

4.3 Evaluación de la solución

ALGUNAS EJECUCIONES QUE MUESTREN VULNERABILIDADES Y SOLUCIONAR ALGUNA

4.4 Resultados obtenidos

Con la información expuesta en el apartado actual queda demostrado que es posible insertar en el proceso de desarrollo de aplicaciones de la compañía un proceso automático de detección de vulnerabilidades que, lejos de perturbar el trabajo de un desarrollador, le proveerá de informes periódicos que le permitan conocer el estado actual de las dependencias que está implementando, así como advertir el equipo de trabajo DevOps de las vulnerabilidades que pudieran contener los contenedores de aplicación utilizados para dar soporte a estas.

5 Conclusiones

Security is not a line in the sand. Protecting your business, customers, citizens' data, should be always your number one priority

Dr. Werner Vogels, 2017
CTO at Amazon.com

El apartado actual, tras haber recorrido en la memoria de este TFM "Seguridad en la integración continua de la metodología ágil y la filosofía DevOps" toda la información relativa al proyecto, está dedicado a presentar los objetivos alcanzados y a presentar las líneas futuras de trabajo en torno al mismo.

El conjunto de aplicaciones aportadas a las herramientas y entorno utilizados surge ante la necesidad de las empresa moderna de vigilar de una manera no intrusiva los posibles agujeros y vulnerabilidades de seguridad que contengan las aplicaciones desarrolladas por ellas y los sistemas donde son desplegados. Esta tarea viene requiriendo un esfuerzo excesivo de parte de aquellas personas que se preocupan por mantener los sistemas lo más protegidos posible.

Uno de los focos de atención en este proceso son las vulnerabilidades en el producto ofertado por la compañía, que pueden estar originadas por la multitud de dependencias de código que va a requerir dicha aplicación para su funcionamiento, así como la inmensa cantidad de imágenes de diferente naturaleza que se despliegan en forma de contenedor en los entornos que dan soporte a las aplicaciones que sustentan el negocio.

Siguiendo esta línea comentada, se decide aportar al proceso de Integración Continua de empresas que ofrecen SaaS un sencillo mecanismo de despliegue de contenedores con un conjunto de herramientas y utilidades que van a permitir poder realizar tareas de análisis estático de manera pasiva y como parte del proceso de trabajo natural de la compañía, analizando las posibles posibles vulnerabilidades en las imágenes de contenedores docker y en las depencias de código utilizado, advirtiendo en el caso de que sean encontradas y aportando informes con los que afrontar la solución al problema.

Por este motivo se puede concluir que, tras el desarrollo de los apartados aquí contenidos, una vez estudiadas las herramientas herramientas aportadas, junto a la posibilidad de despliegue con trabajos automatizados y mecanismo de notificación, se cumple con el objetivo perseguido en la realización de este TFM.

Sin embargo, como es trivial en todo proceso tecnológico, la solución aquí entregada no es estática e inamovible y se presta a ser mejorada, además de a mantener los resultados obtenidos actualizados en el tiempo.

Muchas de las tareas y líneas de avance que este nuevo reto contempla aún son desconocidas, ya que van a surgir en la utilización por parte de los usuarios de esta solución. Otras, en cambio, pueden ser establecidas desde el momento actual. A continuación se presentan algunas de las tareas que conforman la línea futura de trabajo de esta aplicación, identificadas durante la realización del proyecto y la redacción de esta memoria:

- **Aumentar el número de lenguajes de programación contemplados en los análisis estáticos**: existen múltiples herramientas en la actualidad de características similares a las aquí presentadas que cubren un amplio abanico de lenguajes de programación diferentes. Implementar un conjunto mayor de dichas herramientas como parte del conjunto actual es una notable mejora a lo aquí presentado.

- **Preparación del entorno para el despliegue en producción**: Contemplando el escenario en que las comunicaciones se realicen con contraseñas encriptadas y utilizando certificados SSLL

- **Ampliar los repositorios alcanzados**: Aumentando el número de parámetros que los trabajos de Jenkins reciben a la entrada se pueden variar la rama de código a la que se le realiza el análisis, el usuario que almacena el repositorio en su cuenta de GitHub o ambos, ampliando considerablemente el número de posibles repositorios objetivos.

El resultado obtenido a partir del presente TFM es un producto con utilidad tangible, que ya está siendo utilizado en los sistemas de Workshare Inc., proporcionando valor añadido a la actividad en materias de Seguridad de la Información y las Comunicaciones que allí se realiza. Este aspecto es elemento clave a la hora de diferenciar el proyecto realizado, ya que existen múltiples trabajos realizados que nunca llegan a superar la barrera de estudios teóricos, para convertirse en un producto o herramienta final en completo funcionamiento, lo que sin duda es una enorme satisfacción para su autor.

Apéndice A

Repositorio de GitHub

Share your knowledge. It's a way to achieve immortality.

DALAI LAMA
HIGH LAMA OF TIBETAN BUDDHISM

Todo el contenido de la memoria actual para el Trabajo fin de Máster TFM "Seguridad en la integración continua de la metodología ágil y la filosofía DevOps", así como los archivos y la información necesaria para desplegar el entorno de pruebas realizado se encuentran a disposición pública en los repositorios de Github, con el único objetivo de seguir avanzando en el desarrollo de la idea y que permanezca disponible a cualquier usuario que pudiera beneficiarse de la misma. La Figura A.1 muestra el repositorio comentado.

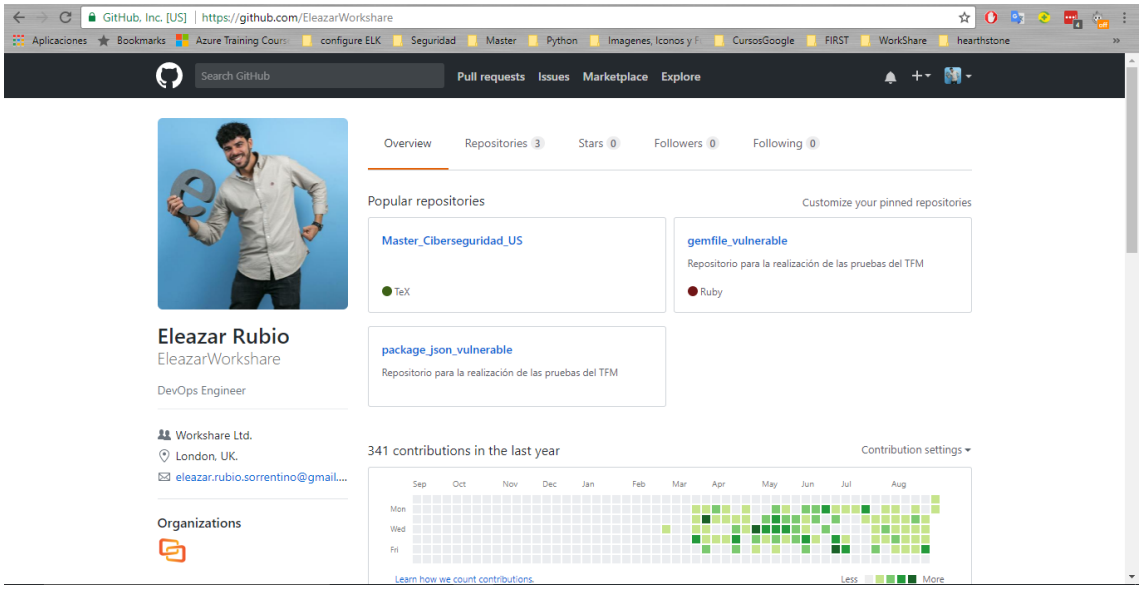

Figura A.1 https://github.com/EleazarWorkshare.

Por supuesto, cualquier contribución será bien recibida.

Índice de Figuras

Índice de Códigos

Bibliografía

[1] Israel Alcázar, *Introducción a Git y Github*, Junio 2014, Enlace.

[2] Elena Arrieta, *DevOps: la tecnología y el negocio deben hablar el mismo idioma*, Mayo 2017, Enlace.

[3] Microsoft Azure, Accedido: Agosto de 2017, Enlace.

[4] Claranet, *DevOps: qué es y cómo lo aplicamos*, Accedido: Agosto de 2017, Enlace.

[5] coreos/clair, Accedido: Septiembre de 2017, Enlace.

[6] CVE, Accedido: Agosto de 2017, Enlace.

[7] Ana M. del Carmen García Oterino, *¿Qué es Docker? ¿Para qué se utiliza? Explicado de forma sencilla*, Julio 2015, Enlace.

[8] Alan R. Earls, *Construir un entorno DevOps con microservicios y contenedores*, Diciembre 2015, Enlace.

[9] Google Cloud Engine, Accedido: Septiembre de 2017, Enlace.

[10] Hewlett Packard Enterprise, *Application Security and DevOps*, Octubre 2016, Enlace.

[11] Docker Inc., Accedido: Septiembre de 2017, Enlace.

[12] _____, *Install Docker*, Accedido: Septiembre de 2017, Enlace.

[13] _____, *Install Docker Compose*, Accedido: Septiembre de 2017, Enlace.

[14] GitHub Inc., Accedido: Agosto de 2017, Enlace.

[15] Consultor IT, *Estudio CA Technologies sobre la importancia de la Agilidad y DevOps en el desarrollo de software [pdf de 20 pgs.]*, Enero 2017, Enlace.

[16] Jenkins, Accedido: Agosto de 2017, Enlace.

[17] Node Security Platform, Accedido: Septiembre de 2017, Enlace.

[18] Adam Prescott, *Automated vulnerability checking with bundler-audit and Travis*, Junio 2015, Enlace.

[19] Erlinis Quintana, *10 razones para usar Github*, Junio 2015, Enlace.

[20] Margaret Rouse, *static analysis (static code analysis)*, Accedido: Septiembre de 2017, Enlace.

[21] rubysec/bundler audit, Accedido: Septiembre de 2017, Enlace.

[22] rubysec/ruby-advisory db, Accedido: Septiembre de 2017, Enlace.

[23] Pedro Santamaría, *Slack, aprende a sacarle partido a la herramienta de comunicación de moda*, Abril 2015, Enlace.

[24] Amazon Wev Services, Accedido: Agosto de 2017, Enlace.

[25] Dependencias software, Accedido: Septiembre de 2017, Enlace.

[26] CA Technologies, *Accelerating Velocity and Customer Value with Agile and DevOps*, Enero 2017, Enlace.

[27] Rubén Velasco, *¿Qué es Docker? ¿Para qué se utiliza? Explicado de forma sencilla*, Febrero 2016, Enlace.

[28] Macarena Vera, *Los beneficios de implementar la Metodología Ágil*, Accedido: Septiembre de 2017, Enlace.

[29] Thought Works, *Continuous Integration*, Accedido: Septiembre de 2017, Enlace.

Glosario

API	Interfaz de Programación de Aplicaciones 15, 24, 25
AWS	Amazon Web Service III, V
CD	Continuous Deployment V
CDS	Ciclo de Desarrollo Seguro III
CI	Continuous Integration V, VII, 3, 15, 21, 30, 41
CVE	Common Vulnerabilities and Exposures III, V, 11, 15
DC	Despliegue Continuo III, VII, 3, 6, 15
DevOps	Development and Operations III, V, 1, 2
GCE	Google Cloud Engine III, V
IC	Integración continua III, VII, 3, 6, 12, 15, 16, 29, 41
OS	Operative System V
QA	Quality Assurance III, V
SaaS	software como servicio III, V, 37
SDLC	Secure Development Lifecycle V, VII, 3, 7
SO	Sistema Operativo III, 3, 8, 10, 13, 15, 19
SW	software III, V, 3, 6, 7, 9, 10
TFM	Trabajo Fin de Máster III, 1–3, 5, 9, 20, 27, 37–39