Chapter 5

Implementación Técnica

En las secciones que conforman este capítulo se ofrece una descripción técnica de cómo se han implementado los códigos que forman parte de la solución final propuesta, desde las imágenes empleadas para instanciar los microservicios y el fichero docker-compose hasta los modelos de aprendizaje automático y profundo con los que se han realizado los análisis.

5.1 Configuración de Contenedores y Microservicios

En primer lugar, antes de abordar los Dockerfiles que definen cada uno de los microservicios y cuyo código puede encontrarse parcialmente en el Anexo correspondiente, conviene entender el funcionamiento de Docker Compose y cómo su fichero correspondiente (*docker-compose.yml*) gestiona la red de dichos microservicios.

El archivo yml es parte fundamental del servicio que se desea desplegar, pues permite levantar, bajar, monitorizar y ejecutar comandos en todos los microservicios empleando una única herramienta de software.

En este fichero se definirán los servicios deplegados otorgándoles un nombre, en el caso aquí contemplado *traffic_gatherer*, *traffic_analyzer* y *results_visualizer*. Además, en el apartado build del sevicio se especifica la ruta al directorio que contiene el fichero Dockerfile que debe emplearse para la construcción de la imagen. Es importante hacer notar que si el Dockerfile hubiese sido creado con un nombre diferente habría que especificar la ruta al propio fichero en lugar de al directorio que lo contiene. Docker Compose construye la imagen cuando se le indica el levantamiento de un servicio si esta no existe en el sistema anfitrión, permite asímismo instanciar la imagen que acaba de crear generando un contenedor operativo cuyo nombre puede especificarse en container_name; se ha optado por nombrar de igual forma al servicio y a su contenedor correspondiente para facilitar al usuario el empleo del sistema.

Tras la información relativa a la imagen y al contenedor se definen dos elementos fundamentales en este caso, por un lado el volumen compartido y por otro el tipo de red.

El volumen compartido es, como se ha mencionado en secciones anteriores, crítico para el funcionamiento del servicio, pues garantiza que los datos y resultados generados en cada uno de los contenedores no desaparecerán si este deja de estar operativo (el problema de persistencia de datos es una de las más importantes limitaciones en Docker y los volúmenes suponen una efectiva solución como se indica en Ibrahim et al. (2021)) y serán accesibles entre microservicios. En este caso se monta como volumen el directorio *shared* que debe ser creado previo despliegue de los microservicios.

El tipo de red es una decisión de enorme importancia a la hora de definir una arquitectura en microservicios. Por defecto, docker compose genera una red interna propia cuando se despliegan los microservicios, sin embargo, como se ha mencionado anteriormente, se ha optado por emplear la red del host para facilitar tanto la comunicación y la exposición de puertos como para permitir a tshark la captura del tráfico de la red de forma sencilla.

A continuación, en el caso del microservicio de captura de tráfico, es necesario especificar su ejecución en modo privilegiado para poder tener permisos de escucha del tráfico. Esto es potencialmente peligroso y motiva aún más la decisión de emplear tokens de autenticación para evitar un uso malintencionado del mismo.

Por último, se indica que el contenedor se reinicie siempre que se detenga para tratar de prevenir microservicios caídos. Esta política de reinicios no se aplica si el contenedor es detenido manualmente pues Docker busca evitar bucles de reinicios.

Dockerfiles

Las plantillas de generación de las imágenes se encuentra definida en los archivos Dockerfiles, estos incluyen los requisitos y los códigos necesarios para el funcionamiento de los microservicios.

Se ha optado en las tres imágenes por partir de una imagen base que incluye Python 3.10 (decidiendo no tomar la versión más reciente de Python y de esta forma garantizar la compatibilidad de todas las librerías) y que corre en un sistema Debian Slim. Esta imagen en Debian Slim garantiza un tamaño extremadamente reducido, consumiendo menos almacenamiento en el dispositivo anfitrión. Aunque existe una forma de disminuir aún más el tamaño de la imagen, empleando como base Linux Alpine, lo minimalista de este sistema hace que haya que instalar a mano de formas poco convencionales ciertas librerías (como podría ser el caso de Pyarrow) y la reducción de tamaño no compensa la complejidad y el tiempo que habría que invertir.

Una vez se ha definido la base, en el caso del *traffic gatherer*, se define un directorio de trabajo y se instala tshark, el software que se empleará para la recolección de datos de tráfico de red. A continuación, se copian los códigos de la aplicación y se realizan las instalaciones necesarias de librerías de python empleando el fichero de requerimientos. Por último, se define el comando de punto de entrada, es decir, aquel que se ejecuta por defecto al instanciar el contenedor, siendo, en este caso, la ejecución de la aplicación FastAPI contenida en su interior y expuesta en el puerto 9000.

DOCKERFILE ANALYZER

DOCKERFILE VISUALIZER

5.2 Captura Automatizada del Tráfico

El microservicio consta de un código principal, *main.py*, en el cual se implementan las medidas de seguridad detalladas en la sección anterior y se define la aplicación FastAPI. Se ha optado por esta herramienta debido a su facilidad de implementación, su baja curva de aprendizaje y por ser, no obstante, una opción robusta que incluye documentación automática de la API generada.

La API consta de una única ruta, /gather, que admite solicitudes HTTP de tipo POST con un parámetro de duración.

Se han empleado librerías como softapi, que permiten la implementación de rate limiters, y la propia librería Python de FastAPI.

El endpoint definido recibe un json en el que solo se espera un parámetro opcional, que representa la duración del tiempo de captura en segundos. Sin embargo, debe proveerse también el token de autenticación como header, pues su verificación es el primer paso en la lógica interna de dicha ruta. Siguiendo, en caso de una correcta autenticación, la llamada a la función principal.

La captura de tráfico se realiza mediante un comando tshark en el que se le especifica la red local como objetivo de captura y la ruta al fichero PCAP donde se deberá escribir el resultado. Este archivo generado será almacenado en el volumen compartido. Se analiza el código de respuesta de tshark para evalúar si la recogida ha sido correcta o no y poder monitorizar el error.

Se ejecuta entonces nuevamente tshark para extraer la información relevante sobre los envíos del PCAP y almacenarla en un fichero CSV para su posterior procesado y análisis, el cual se realiza a continuación, agrupando los envíos en flujos en base al emisor, el receptor y el protocolo y almacenándolo en otro archivo CSV. Es importante que estos resultados generados se guarden también en el volumen compartido, de esta forma el microservicio de análisis podrá hacer uso de ellos.

5.3 Preprocesamiento

La modularidad, la independencia y la reutilización son principios fundamentales que guían este proyecto, es por ello que la concepción de la etapa de preprocesamiento se ha hecho desde una clase DataProcessor para posibilitar su despliegue en cualquier contexto y evitar la repetición de código innecesario.

El primer paso en el proceso de preprocesado es la carga del dataset, en este caso un archivo CSV empleando pandas. Al haberse trabajado con datos de tráfico real el siguiente paso, si no obligatorio, es altamente recomendable; realizar limpieza de los datos.

A la hora de limpiar los datos con los que se van a realizar predicciones empleando modelos hay dos decisiones importantes que se deben tomar, la gestión de valores duplicados y la de datos faltantes. En este caso, como se argumentará más adelante cuando se presenten las pruebas y validaciones realizadas, se ha optado por eliminar los registros duplicados y sustituir aquellos valores numéricos faltantes por cero.

Una vez el dataset se encuentra limpio se procede a normalizar los datos. En este caso aplicando un StandardScaler. Normalizar la escala de datos de una variable facilita el uso de determinados modelos, especialmente aquellos que dependen de distancias o pesos, como las redes neuronales. La normalización evita que variables que están en una escala numérica muy diferente a otras puedan dominar en la elección de estos pesos o distancias.

A continuación, para el uso de algunos modelos, conviene convertir en identificadores numéricos las variables categóricas, pues estos no admiten datos no numéricos.

La última parte del preprocesado, aunque no parte del despliegue final en producción, forma parte del servicio y ha resultado fundamental en la parte de validación y comparativa de modelos.

Por un lado, los datos de tráfico de una red suelen presentar un desbalanceo de clases muy acuciado, siendo muy predominante el tráfico normal. La detección mediante aprendizaje automático se fundamenta en poder entender los patrones subyacentes en el tráfico malicioso y en el normal, por tanto, la existencia de muy pocos registros de botnet dificulta este trabajo. Es por ello, que se ha decidido, para el entrenamiento de los modelos, realizar balanceo de clases en el conjunto de entrenamiento. Entre las técnicas seleccionadas para esto se ha optado por aplicar tanto SMOTE como Tomek Links y comparar resultados.