Chapter 4

Diseño del Servicio

El objetivo final del presente trabajo es el diseño y posterior despliegue de un servicio de análisis de tráfico y detección de Botnets integrado en una red IoT. Es por ello que una de las primeras consideraciones debe ser el enfoque arquitectónico del mismo; el diseño de una topología adecuada es fundamental para lograr un servicio lo más eficiente posible.

4.1 Requisitos

Se han establecido diversas prioridades en el diseño del sistema de detección, las cuales pueden ser categorizadas como funcionales o no funcionales.

Los requisitos funcionales son 3: captura de tráfico bajo demanda, análisis de dicho tráfico y presentación de resultados.

El servicio puede considerarse un sistema de seguridad integrado en la red que, o bien periódicamente o bien cuando el responsable de la misma detecta alguna anomalía, pueda ser consultado. Por tanto, debe existir un endpoint expuesto en el cual se puedan realizar peticiones que inicien el análisis. La duración de la captura será configurable por la persona que realiza la solicitud, capturando todo el tráfico de la red durante el periodo de tiempo establecido y generando al terminar un archivo de tipo pcapg apropiado que podrá emplear el proceso de análisis de tráfico.

El análisis de tráfico, como se describirá más adelante, consiste en un preprocesamiento de los datos capturados en el fichero para hacer posible su ingestión por parte de un modelo de aprendizaje automático. Este modelo habrá sido previamente entrenado y será el mejor de todos aquellos probados en la fase de testing. Este debe ser capaz de producir como resultado un análisis que determine la probabilidad de que en el tráfico que ha analizado haya presente una botnet.

Sin embargo, conviene tener en cuenta otro tipo de cuestiones de diseño, aquellas reltivas a la eficiencia y a la interoperabilidad.

Un buen servicio o aplicación pensado para desplegarse en un entorno de Internet de las Cosas, heterogéneo tanto en software como en hardware por su propia naturaleza, debería ser capaz de funcionar en prácticamente cualquier sistema o dispositivo, es decir, *independencia de la plataforma*. Esta interoperabilidad debe garantizarse a todos los niveles, tanto en requerimientos del sistema operativo como en protocolos.

Además, presentando 3 funcionalidades claramente diferenciadas (captura, análisis y visualización) resulta interesante apostar por la modularidad en su arquitectura, haciendo que estas 3 funciones puedan desplegarse y ejecutarse de forma independiente, pudiendo

así utilizar solo una o dos de ellas sin que se requieran ajustes.

Por último, resulta fundamental la seguridad y la tolerancia a fallos. Poder actuar ante la caída de parte del servicio y, además, garantizar el acceso al mismo solo a las personas apropiadas (siendo una aplicación con acceso interno al tráfico de la red) son dificultades de primer nivel en el despliegue.

Tras estudiar los requisitos anteriormente mencionados se ha considerado la arquitectura containerizada en microservicios como el diseño más apropiado para este sistema de detección de botnets. Un enfoque basado en microservicios permite el despliegue de aplicaciones independientes y el uso de contenedores facilita su portabilidad y modularidad. Además, garantiza el funcionamiento del sistema si una de las funcionalidades caen, pudiendo, por ejemplo, seguir capturando tráfico y almacenar el fichero resultante incluso si el sistema de análisis está momentáneamente inoperativo y es empleado más tarde. Esta robustez también facilita el trabajo de monitoreo y parcheado del sistema pues, a diferencia de una aplicación monolítica tradicional, permite la modificación y el redespliegue de parte en lugar de todo el servicio, agilizando el trabajo.

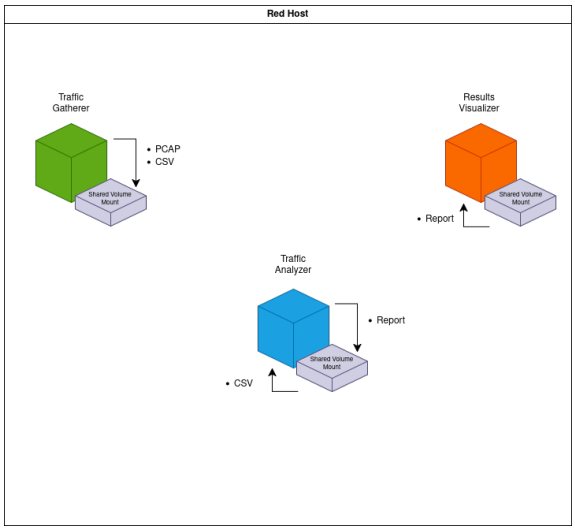

Figure 4.1: Implementación de la Arquitectura del servicio

4.2 Arquitectura general del sistema

4.2.1 Diseño mediante microservicios

El primer microservicio que actúa en la pipeline es **Traffic Gatherer**. Su objetivo es capturar el tráfico de la red bajo demanda y almacenarlo de forma apropiada para su posterior análisis.

Se abordará este microservicio desde tres enfoques diferentes; técnico, funcional y de seguridad. A nivel técnico el proceso de recolecta de tráfico se basa en FastApi, un framework sencillo, rápido y robusto para construir y desplegar APIs en Python. Mediante esta herramienta se expone un endpoint que permite iniciar el microservicio con una sencilla solicitud HTTP. El primer paso es inicializar tshark para comenzar la recogida de tráfico durante un periodo determinado de tiempo (configurable en la solicitud) y generar un pcap con los resultados. Posteriormente se emplea la misma herramienta, la mencionada tshark, para convertir el pcap en un archivo csv con la información de interés para el análisis (en este caso la marca de tiempo, las direcciones ip de origen y destino, el protocolo empleado,

y los bytes del paquete). Por último, se analizan estos envíos para generar flujos a partir de ellos, considerando parte de un mismo flujo aquellos paquetes enviados por la misma ip con destino cada una de las otras ips y empleando siempre el mismo protocolo. Estos 3 ficheros generados (el archivo pcap, el archivo crudo de paquetes y el archivo de flujos) son almacenados en un volumen montado compartido por todos los microservicios del sistema.

Desde un punto de vista funcional, como se ha mencionado anteriormente, el servicio expone un endpoint (*gather*) que admite solicitudes HTTP de tipo POST. En estas solicitudes se admite un parámetro opcional *duration* que especifica la duración de la captura del tráfico, siendo por defecto de una hora. Esto se ha parametrizado en lugar de fijar un tiempo para todas las solicitudes con el objetivo de permitir al usuario más flexibilidad sobre todo de cara al análisis posterior (puede optar por una captura más corta para un análisis más rápido o permitir a tshark recolectar datos durante mucho tiempo para hacer un análisis más robusto del tráfico). El resto del proceso ocurre de forma independiente al usuario, es decir, la persona que ha generado la solicitud de recogida de tráfico no tiene que ir solicitando la generación de los ficheros ni de los flujos conversacionales, el propio programa realiza todo de forma secuencial. Por último, y si se deseara poder observar los resultados, estos podrían encontrarse en el directorio compartido *shared*.

Bajo un punto de vista de seguridad se ha considerado fundamental garantizar dos características. Por un lado, proteger el servicio de posibles ataques de Denegación de Servicio por inundación de solicitudes (HTTP Flood) y, por otro, no permitir peticiones de personas no autorizadas.

Con el fin de evitar el primero de los problemas mencionados se ha optado por implementar un sencillo sistema de *rate limiting*. Es decir, se establece un límite al número de solicitudes en un periodo de tiempo, en este caso una por minuto. Si se recibiera más de una solicitud por minuto el sistema bloquearía y no gestionaría las solicitudes posteriores; de esta forma se evita la capacidad de un posible atacante de saturar el servidor.

En cuanto al problema relativo a la autorización se empleará un mecanismo de Token. Una función verifica si, como parte de los headers de la solicitud, se ha adjuntado un token que coincide con el almacenado en el archivo de variables de entorno. Si coincide entonces se procesa la solicitud, en caso contrario se deniega por error en la autenticación.

El siguiente microservicio creado corresponde al **Traffic Analyzer**. Este servicio representa una papel fundamental en el proceso pues es el que se encarga de emplear un modelo de aprendizaje automático que ha sido previamente entrenado para determinar la probabilidad de que se esté sufriendo un ataque mediante una Botnet. De la misma forma que el anterior se abordará desde un punto de vista técnico, funcional y de seguridad, aunque muchas implementaciones serán equivalentes a las explicadas en el microservicio anterior.

A nivel técnico el módulo carga un modelo de aprendizaje automático que ha sido previamente almacenado como archivo serializado mediante pickle. Este modelo ha sido ya entrenado previamente y, por tanto, se encuentra ya preparado para realizar las predicciones. Para ello, se carga el dataset que se encuentra en el volumen compartido como archivo csv y se realizar un preprocesado inicial, codificando las variables categóricas, normalizando aquellas que son numéricas y prescindiendo de características que no aportan la suficiente información si es necesario. Posteriormente el modelo realiza su predicción y esta se almacena, junto a algunos datos relativos al análisis, en un fichero csv también ubicado en el volumen compartido.

Desde un punto de vista funcional, de forma semejante al microservicio de recogida, se expone un endpoint que recibirá solicitudes HTTP de tipo POST para dar inicio al análisis. No requiere mayor interacción por parte del usuario, pudiendo este observar el

análisis en el el directorio compartido *shared*.

Por último, a nivel de seguridad, se han empleado las mismas técnicas de protección que en el microservicio previamente explicado. Por un lado , se requerirá la autorización del usuario que realiza la petición mediante un token de acceso, admitiendo la solicitud si este es correcto y denegándola si no lo es. Por otro lado, se limita la cantidad de peticiones posibles al servicio para un intervalo de tiempo empleando *rate limiting*.

FALTA VISUALIZADOR

4.2.2 Despliegue con Docker Compose

Como se ha dicho previamente, se ha optado por emplear Docker Compose como herramienta de orquestación de los diferentes microservicios.

En este caso, se instanciarán 3 contenedores, uno por microservicio. Los 3 contenedores no estarán desplegados en una red propia y aislada del sistema anfitrión, si no que compartirán la red host. Esto se realiza para poder capturar fácilmente mediante tshark el tráfico de la red a la que pertenece el sistema que aloja el servicio y para exponer los microservicios en los propios puertos del host.

Surge además un problema clásico en las arquitecturas modulares que buscan implementar microservicios completamente independientes, poder hacer uso en todos los contenedores de ficheros generados por el resto. Para ello se monta un volumen (un almacenamiento persistente de los datos) común en los tres. Este volumen será quien aloje los ficheros generados por cada uno de los procesos para que el resto pueda emplearlos. También como parte del conflicto entre interdependencia y practicidad se ha presentado otra dificultad; los microservicios no deben llamarse entre ellos directamente de forma secuencial. Si esto fuera así se facilitaría mucho la implementación del proceso y la necesidad de interacción del usuario sería mínima, pues este podría solicitar al primer microservicio el inicio del proceso completo y este se encargaría, tras recolectar datos, de llamar al proceso de análisis, quien a su vez iniciaría el microservicio de visualización. Sin embargo, esto choca frontalmente con la modularidad y la independencia, pues los microservicios fallarían si se implementaran o desplegaran parcialmente (por ejemplo, si solo traffic gatherer se hubiera desplegado este fallaría pues, al acabar, trataría de llamar al contenedor de análisis que no se encontraría en el sistema). Para evitar esta situación, y tras barajar opciones como emplear un sistema de colas como RabbitMQ o realizar publicaciones empleando, por ejemplo, mosquitto, y que el microservicio siguiente se activara mediante escucha de dicha publicación, se ha optado por una opción más sencilla; cada microservicio es invocado de forma independiente exponiendo un endpoint. Esta solución garantiza la independencia de los servicios, no complica en exceso el sistema y permite realizar funcionalidades parciales como, por ejemplo, solo recoger tráfico sin analizarlo.

Este despliegue y orquestación es reproducible empleando el fichero docker-compose.yml, que define cada uno de los servicios y su forma de despliegue.