Universidad de Oviedo

ESCUELA POLITÉCNICA DE INGENIERÍA DE GIJÓN.

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

ÁREA DE INGENIERÍA TELEMÁTICA

APLICACIÓN DE SEGURIDAD PERIMETRAL EN REDES DEFINIDAS POR SOFTWARE

D. Carlos Tristán González Álvarez
TUTOR: D. MELENDI PALACIO, David
COTUTOR: D. NUÑO HUERGO, Pelayo

FECHA: Julio 2020

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

Acrónimos

ARP ..... Protocolo de resolución de direcciones.

CLI ..... Interfaz por línea de comandos.

CPU ..... Unidad central de procesamiento.

DNS ..... Sistema de nombres de dominio.

DoS ..... Denegación de Servicio.

GB ..... GigaByte.

HTTP ..... Protocolo de transferencia de hipertexto.

HTTPS ..... Protocolo de transferencia de hipertexto seguro.

ICMP ..... Protocolo de mensajes de control de Internet.

IETF ..... Grupo de trabajo de ingeniería de Internet.

LLDP ..... Protocolo de descubrimiento de la capa de enlace.

LTS ..... Soporte de largo plazo.

MV ó VM ... Máquina Virtual.

NAT ..... Protocolo Traductor de direcciones de red.

RAM ..... Memoria de acceso aleatorio.

SDN ..... Red definida por software.

SNMP ..... Protocolo Simple de configuración de red.

SSH ..... Intérprete de órdenes seguro.

SSL ..... Capa de conexión segura.

TCP ..... Protocolo de Control de transmisión.

TLS ..... Seguridad en capa de transporte.

XML ..... Lenguaje de marcas extensible.

Agradecimientos

En primer lugar a mis tutores, David y Pelayo, por ofrecerme la oportunidad de este trabajo, del que he aprendido mucho. Por toda su ayuda, su atención y gran profesionalidad.

A todos mis compañeros de máster durante estos últimos dos años. Muy especialmente a D. Enrique Álvarez Cofiño-García, D. Alejandro Casanova Alonso y D. Javier Montes Antuña. Por todas las ayudas que me han ofrecido, su gran compañerismo y, ante todo, grandes personas tanto dentro como fuera del ámbito de estudios. Muchas gracias.

A mi familia, por apoyarme en todo momento.

A todo el profesorado del máster, coordinadores e integrantes de la comisión académica. Por su afán constante de mejora, dedicación, esfuerzo y entrega en este gran máster.

En Avilés, a 8 de julio de 2020.

1. Introducción

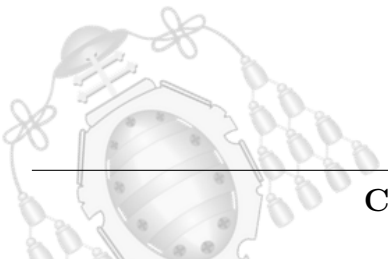
En este capítulo se comentan los principales objetivos del proyecto, así como las motivaciones para llevarlo a cabo y el ámbito de estudio de este trabajo.

1.1.- Objetivos

El propósito de este trabajo es el estudio, investigación y desarrollo de aplicaciones de seguridad perimetral para Redes Definidas por Software (SDN). Este tipo de medidas en redes convencionales requieren, por lo general, de hardware específico y una correcta configuración de los dispositivos de la red, lo que supone un incremento del coste y los recursos necesarios para ello.

Son muchos los ataques que pueden realizarse en un entorno LAN. De todos ellos, se van a investigar algunos de los más extendidos, como por ejemplo el escaneo de servicios, en donde un usuario malintencionado intenta obtener información de todos los dispositivos de la red, con el objetivo de poder explotar sus vulnerabilidades. También son objeto de estudio ataques unidireccionales en redes VLAN y PVLAN, en los que el atacante puede enviar tráfico ilegítimo a usuarios a los que no tiene permitido el acceso. También se investiga la forma de detener ataques para el envenenamiento de rutas, donde un usuario malicioso modifica el comportamiento de la red con la intención de interponerse en las comunicaciones y capturar el tráfico entre dos o más equipos pudiendo obtener información sensible como contraseñas o datos bancarios, entre otros.

Es por ello que se aprovecha la versatilidad de las redes SDN así como su programabilidad para conseguir los mismos resultados que con las técnicas implementadas en las redes tradicionales pero reduciendo el coste y el esfuerzo.



1.2.- Motivación

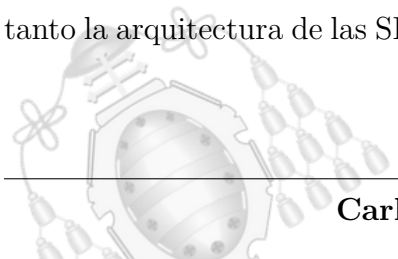
Las Redes Definidas por Software constituyen hoy en día un hecho en muchos aspectos del mundo de las redes: centros de datos, redes cloud o algunos ámbitos universitarios son la prueba de que este nuevo paradigma, nacido en 2009, está cada día más presente y ha venido para quedarse.

Una de las partes que componen este tipo de redes es la capa de aplicaciones: piezas o conjuntos de software que interactúan con la red y pueden modificar su comportamiento. Existen multitud de ellas enfocadas a diversos propósitos: encaminamiento de tráfico, políticas de calidad de servicio o seguridad. Las aplicaciones pertenecientes al ámbito de la seguridad, en la actualidad, son escasas y dejan al descubierto muchas carencias solucionadas en el concepto tradicional de redes por medio de hardware específico o la aplicación de varios protocolos simultáneamente. La llegada de las SDN ha mejorado algunos aspectos respecto a las redes tradicionales, sin embargo en el aspecto de seguridad los problemas se mantienen e incluso aparecen algunos nuevos.

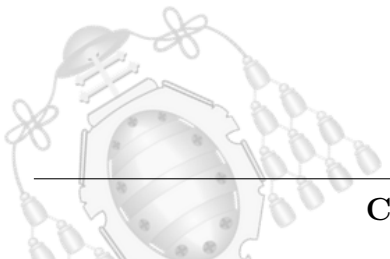
La seguridad de las redes de comunicación es un requerimiento indispensable en nuestros días, máxime con situaciones como las actuales -a fecha de realización de este trabajo- de crisis sanitaria en la que su correcto funcionamiento es un aspecto crítico. La posibilidad que brindan las SDN de una programación *ad hoc* de su comportamiento hacen de ello un concepto muy interesante para su explotación en el ámbito de la seguridad perimetral.

1.3.- Ámbito de estudio

El ámbito de estudio de este proyecto se centra en la capa de aplicaciones de las Redes Definidas por Software. Esta capa permite modificar el comportamiento de los dispositivos de la red y aprovecharlo para mitigar ataques que suceden tanto en las redes convencionales como en las definidas por software. Para su desarrollo debe conocerse tanto la arquitectura de las SDN como la anatomía de los ataques que se van a estudiar.



También es objeto de trabajo la capa de control, que aglutina toda la lógica de las SDN y da soporte a las aplicaciones anteriormente mencionadas.



2. Redes definidas por software

Antes de dar paso al estudio y desarrollo de aplicaciones de seguridad en este nuevo concepto de red, es preciso dar a conocer ciertos detalles: qué son las SDN, los elementos que la conforman, sus precedentes o qué nuevos protocolos llevan asociados, entre otras cuestiones.

2.1.- Definición

Las redes definidas por software constituyen un nuevo concepto de arquitectura de red, que separa el plano de datos del plano lógico de control, unificando este último en un solo elemento externo a la red física, conocido como controlador o sistema operativo de red.

A diferencia de las redes tradicionales, la separación entre estos dos planos es la pieza clave a la hora de alcanzar mayor flexibilidad, reducir las tareas de gestión y mantenimiento y convertir las redes en sistemas mucho más abiertos a nuevos cambios y mejoras.

El controlador es el elemento que contenga toda la lógica de la red. Cuenta con una visión general de su topología y, junto con otra información, se encarga de elaborar, y posteriormente enviar a cada dispositivo (switches y routers) las reglas de reenvío que deben seguir. También tiene la capacidad de controlar y gestionar todos y cada uno de los dispositivos de la red, como cortafuegos, servidores DNS, servidores NAT, etcétera.

Los cuatro pilares [1] que definen este nuevo concepto de red son:

- Separación del plano de datos del plano de control.
- El reenvío de paquetes por parte de los encaminadores o enrutadores se hace en base a las reglas establecidas por el controlador.
- El controlador es el elemento fundamental.

- Son redes programables. Por medio de la construcción de software, es posible el manejo del plano de datos.

La arquitectura SDN se divide en tres capas bien diferenciadas, ilustradas en la figura 2.1 y que se explican a continuación: Control, Aplicación e Infraestructura.

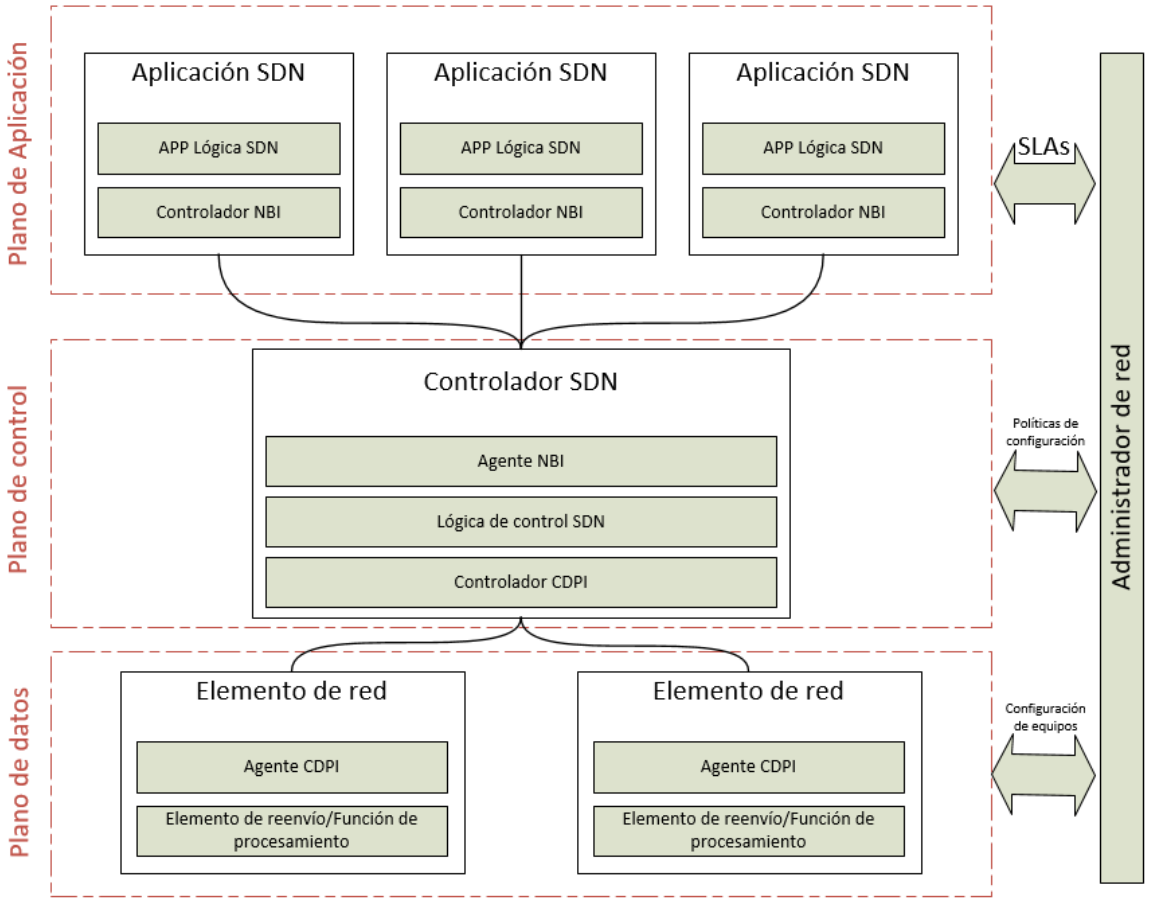


Figura 2.1.- Vista general de la arquitectura de las redes definidas por software [2]

2.2.- Tecnologías previas

Las redes SDN son el fruto de mejoras en modelos preexistentes que no tuvieron el impacto y el apoyo suficiente. Desde la aparición de Internet, elementos como los enrutadores o los conmutadores han ido incrementando progresivamente la integración de nuevos servicios y mejoras en la seguridad. Paralelamente, la capacidad de computación ha ido aumentando casi exponencialmente su valor.

Estos dos tipos de evoluciones han contribuido de manera especial en el surgimiento de la idea de separar el plano de control del plano de reenvío o datos.

Una de las primeras tecnologías es la denominada ForCES [3] (*Forwarding and Control Element Separation*). Desarrollada a finales de los años 90 por el grupo de trabajo IETF, distingue la existencia de dos entidades lógicas:

- **Elemento de control (CE)**: Implementa el protocolo ForCES e indica a los elementos de reenvío las directrices de manejo de paquetes.
- **Elemento de reenvío (FE)**: Implementa el protocolo ForCES y su misión es obedecer las órdenes enviadas por un elemento de control (CE).

Así mismo, dentro de cada capa del protocolo, existía un *manager*. En el caso de la capa de control, se encargaba de determinar qué elementos de control estarían activos durante las fases de arranque del sistema. Para la capa de reenvío o datos, asocia cada elemento de reenvío a uno o varios elementos de control.

Entre otras características, destacan su escalabilidad con soporte para cientos de elementos de reenvío en una misma red y la posibilidad de utilizar varios elementos de control, obteniendo una red redundante y tolerante a fallos.

La falta de estandarización de este protocolo, y el bajo interés de los fabricantes, hicieron que su puesta en marcha supusiese más una utopía que una realidad.

Otro proyecto, que sería previo a la aparición de las SDN, tenía el nombre de Ethane [4]. El creciente número de operaciones que un dispositivo de red podía ejecutar, unido a una mayor sofisticación de las mismas, incrementaban el nivel de complejidad traduciéndose, en definitiva, en una menor escalabilidad. La solución pasaba por delegar todas estas tareas a un dispositivo de orden superior, actuando el resto como simples encaminadores de paquetes.

A diferencia de ForCES, su arquitectura es mucho más simple y muy similar a la de las SDN: Un controlador de red que decidía las reglas de envío y unos encaminadores cuya única misión era reenviar los paquetes.

Las dos principales características de una red Ethane eran:

- Existía total control sobre quién enviaba cada paquete dentro de la red. Cada usuario estaba registrado dentro de la red, así como su ubicación y acciones.
- Una política de encaminamiento centralizada en un único elemento físico, el controlador, el cual decidía quién podía comunicarse con quién. Esto era posible añadiendo reglas de encaminamiento en cada dispositivo. Si no existía ninguna regla coincidente, el paquete era descartado directamente.

Su adopción fue algo más significativa que ForCES, puesto que tenía mayor compatibilidad con los dispositivos presentes en el mercado. Es considerado el predecesor de protocolos como OpenFlow, el pilar fundamental de las SDN.

2.3.- Arquitectura

En los siguientes apartados se procede a explicar cada una de las capas que conforman este nuevo concepto de redes. Existen tres bien diferenciadas: capa de Aplicación, capa de Infraestructura y capa de Control.

2.3.1.- Capa de Aplicación

La capa de aplicación aúna a todos los programas que comunican sus comportamientos y recursos con el controlador de la red. Esta comunicación controlador-aplicaciones se realiza mediante las denominadas «Interfaces de aplicación del programa» - APIs, en inglés -, es decir, un software intermediario que permite a dos o más aplicaciones hablar, bien entre ellas o hacia otro elemento. Estos programas pueden construir una visión abstracta de la red gracias a la recolección de información proveniente del controlador

Existen aplicaciones centradas en la mejora del enrutamiento de paquetes, gracias al uso de técnicas de balanceo de carga, en caso de redes muy congestionadas. Otras están más focalizadas en mejorar la experiencia del usuario (QoS), o el ámbito de la seguridad. Algunos ejemplos concretos son QNOX (QoS-Aware Netwotk Operating

System) orientado a la calidad del servicio o MonSamp, diseñada para monitorizar flujos de tráfico.

Opciones dedicadas al mantenimiento y mejorar la seguridad de la red también están disponibles, favorecidas en buena parte por la amplia visión de la red de la que se dispone. Programas automatizados como OFRewind [5], ndb o NetSight son un claro ejemplo de sistemas de auditoría de red basadas en eventos. Otras como SFlow-RT [6], analizan los flujos de tráfico entrante al controlador y actúa de forma pasiva ante posibles ataques por denegación de servicio.

En el plano de las redes definidas por software, se denomina el *Northbound* a las comunicaciones entre controlador de red y aplicaciones. Es por ello que se definen como «Interfaces NourthBound» al conjunto de APIs que componen esta parte (NorthBound Interfaces, NBIs)

Dependiendo del tipo de controlador que se utilice, existen diversos tipos de aplicaciones. En lo que respecta al ámbito de la seguridad, son varias las aplicaciones encomendadas para esta cuestión [7]: Desde ataques por denegación de servicio (DoS) hasta defensa frente a mutaciones de equipos aleatorias, como OF-RHM.

El desarrollo de estas aplicaciones depende en buena medida del soporte que provea el controlador. Cada uno consta de sus propias librerías. El más extendido es Java, presente en todas las versiones los controladores más relevantes. Así mismo, existen otras alternativas tanto en C como en Python.

2.3.2.- Capa de Control

Es la más importante de todas, ya que concentra toda la lógica e inteligencia de la red. El elemento principal es el controlador y sirve de nexo entre los dispositivos de la red y las aplicaciones.

La comunicación con los elementos de reenvío de la capa de infraestructura se realiza mediante Interfaces SouthBound del plano de control (CPSI en inglés) que son los equivalentes a las APIs utilizadas para la comunicación entre la capa de control y la

de aplicación. Gracias a esta interacción con la capa de infraestructura, el controlador es capaz de extraer información sobre la red y el hardware de los dispositivos, incluyendo estadísticas y eventos sobre lo que está pasando en todo momento.

En función de los requerimientos de las aplicaciones, el plano de control actúa sobre los distintos elementos que conforman la capa de infraestructura, por ejemplo mediante la actualización de las reglas de reenvío, bien sea para una rápida recuperación tras un fallo, balanceo de carga de tráfico sobre un determinado segmento de la red o aplicación de políticas de seguridad.

Esta capa es, sin duda, el objetivo principal de un ataque cibernético. Un fallo de seguridad podría comprometer a toda la estructura de una SDN puesto que la intrusión de agentes maliciosos no solo les permitiría tener acceso al controlador, sino también tanto al plano de aplicaciones (a través del *NorthBound*) como el plano de infraestructura (a través del *SouthBound*). Puede suponer también la captación de tráfico entre el controlador y el resto de capas de la red, llegando incluso a la modificación de los contenidos de las tablas de enrutamiento de sus dispositivos.

Por último, destacar que, para hacer posible la comunicación entre la capa de control y las aplicaciones, es necesario realizar un proceso de *traducción* de los requisitos de estas últimas a un lenguaje de programación de más alto nivel. Las alternativas disponibles van desde los más conocidos como Python, C++ o Java, soluciones intermedias con el uso de librerías como Cisco OnePK ó, por último, nuevos lenguajes específicamente diseñados para ello tales como FML (Flow-based Management Language) o Nettle.

2.3.3.- Capa de Infraestructura

Esta capa comprende todos los elementos hardware que conforman la red, los cuales comparten todas sus capacidades hacia la capa de control, mediante interfaces «SouthBound». Estos dispositivos realizan, además del envío y recepción de paquetes, dos tareas distintas: control y gestión. En el caso del control, el hardware de red se encarga de recolectar el estado de la misma (tráfico, congestiones, topología a través

del protocolo LLDP, etcétera) y ésta es enviada al controlador, quien tras recibir esta información elabora las reglas necesarias de reenvío de paquetes.

En el caso de la gestión, se debe focalizar la diferencia de las SDN con respecto a las redes tradicionales, en donde en estas últimas los dispositivos, de manera autónoma, ejecutan complejos algoritmos de encaminamiento para obtener la mejor ruta. Para las redes definidas por software esta tarea es delegada en el controlador. En caso de que un paquete no coincida con ninguna de las reglas, los dispositivos de la red envían ese paquete al controlador para su posterior procesamiento. De esta manera, los dispositivos de red se encuentran en un aprendizaje constante a medida que rellenan de información sus tablas de rutas.

2.4.- El protocolo OpenFlow

OpenFlow y SDN son términos que en múltiples ocasiones se utilizan en la misma frase, pero que para nada significan lo mismo. Mientras SDN es la arquitectura de la red, OpenFlow define la forma en que el controlador se comunica con los dispositivos.

Gestionado por la Open Network Foundation (ONF) desde marzo de 2011, vio la luz en su primera versión (0.1.0) en noviembre de 2007 fruto del trabajo de unos desarrolladores de la Universidad de Standford. La primera versión estable (1.0.0) y, a su vez, la más ampliamente conocida, no llegó hasta diciembre de 2009. A día de hoy, la versión más reciente es la 1.5.1, publicada en abril del año 2015. Se tiene constancia de la existencia de una versión 1.6 que data del año 2016, aunque la especificación de la misma aún no ha sido puesta en conocimiento por parte de esta organización.

Se trata de un protocolo abierto y que está diseñado para manejar y dirigir el tráfico de una red de forma centralizada, independientemente de su heterogeneidad.

OpenFlow aprovecha la existencia, en la gran mayoría de los dispositivos Ethernet, de tablas de flujos (generalmente usadas para funciones de cortafuegos, parámetros de calidad de servicio o estadísticas de tráfico). Estas tablas, aún siendo diferentes en función del fabricante, comparten una serie de características comunes. Un dispositivo OpenFlow consta de, al menos, tres partes bien diferenciadas:

- Tabla de flujos o tabla «flow», que contiene todas las reglas de intercambio de paquetes. El dispositivo busca alguna coincidencia en ellas cada vez que le llegue un paquete.
- Canal de comunicaciones, a través del cual el dispositivo se conecta al controlador de la red.
- El propio protocolo OpenFlow, que establece las reglas de intercambio de mensajes entre controlador y dispositivo, así como el manejo de sus tablas de flujo (añadir nuevas rutas, eliminarlas o modificarlas).

En la figura 2.2 se muestra un pequeño esquema de la composición de un switch OpenFlow y cómo éste se conecta por medio de un canal al controlador.

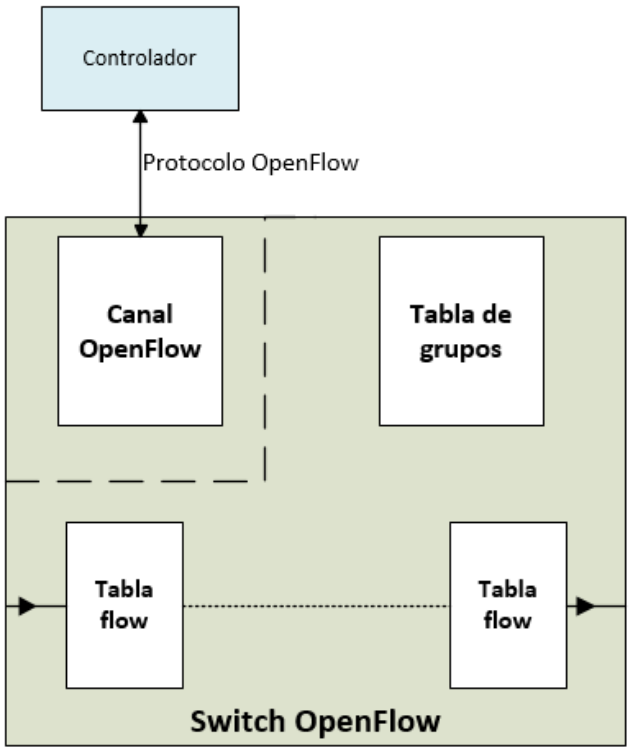


Figura 2.2.- Esquema de la composición de un switch OpenFlow [18].

2.4.1.- Detalles del protocolo

OpenFlow cuenta un sencillo vocabulario de mensajes y sus propias reglas de intercambio. Todos los mensajes cuentan con una cabecera común, en la que se incluye

información como la versión del protocolo y el tipo de mensaje, tal y como se puede observar en la figura 2.3. Estos mensajes pueden dividirse en tres grandes grupos:

- De controlador a Switch.
- Asíncronos.
- Simétricos.

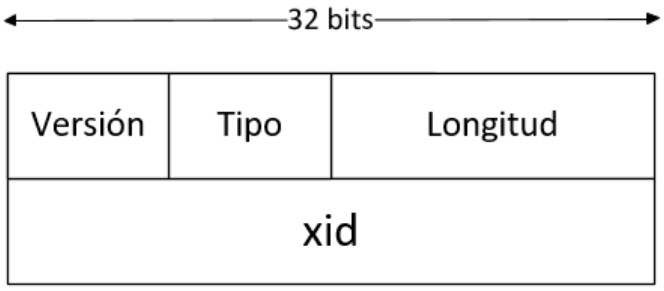


Figura 2.3.- Composición de la cabecera de un paquete OpenFlow [19].

2.4.1.1.- Mensajes de controlador a switch

Son inicializados por el controlador de la red y pueden requerir o no una respuesta. Entre los que la requieren, están los llamados **Read-state** para obtener estadísticas de un dispositivo, o los mensajes **Feature**, cuya respuesta son las características que soporta el elemento de red al que se preguntó.

Mensajes como **Modify-State**, usados para modificar ciertos parámetros de los switches de la red o alguna de las entradas de sus tablas de flujo no requieren respuesta alguna.

2.4.1.2.- Mensajes asíncronos

A diferencia de los anteriores, son enviados desde los elementos de red hacia el controlador, sin que éste haya realizado ninguna petición previa. La especificación de OpenFlow habla de cuatro principales tipos de mensajes asíncronos. **Packet-in**, cuando un paquete no tiene ninguna entrada en la «Tabla flow» del elemento de la red; **Flow-Removed**, si una entrada de la «Tabla Flow» ha sido eliminada; **Port-Status**, si ha

cambiado el estado de algún puerto; **Error** que, como su propio nombre indica, avisa de si se ha producido algún tipo de error.

2.4.1.3.- Mensajes simétricos

Son mensajes bidireccionales: De controlador a switch y de switch a controlador. Un ejemplo es el mensaje de **Hello**, intercambiado entre controlador y switch a la hora de inicializar la conexión. Otro ejemplo, es el mensaje de **echo**: puede ser enviado desde cualquier tipo de elemento y debe ser respondido.

2.4.2.- Funcionamiento básico

En los siguientes apartados se expone el funcionamiento de una red SDN desde su arranque hasta su fase de funcionamiento normal.

2.4.2.1.- Fase de arranque

Todo comienza con el establecimiento de conexión entre el controlador de red y el switch. Esta se realiza a través de una dirección IP fijada con anterioridad, conocida por el dispositivo de red y en el puerto TCP número 6633. Esta conexión puede estar asegurada mediante el protocolo TLS, con el fin de hacer las comunicaciones más seguras, aunque este tema se trata con más detalle en futuros apartados. Como se puede deducir, este establecimiento es iniciado por el switch o router de la red. Los primeros mensajes que se intercambian, son mensajes simétricos.

El primero de todos que se intercambian es «Hello» un mensaje consistente únicamente en una cabecera que incluye, entre otros datos, la versión del protocolo que es capaz de soportar cada máquina que interviene en el proceso. Esto es necesario con el fin de que no existan problemas a la hora de realizar la comunicación. Siempre se efectúa utilizando la versión más baja de las dos. En caso de que controlador o elementos de red no puedan trabajar con la versión del otro, se envía un mensaje asíncrono de tipo «Error» que contiene los detalles del mismo y la conexión se da por finalizada.

2.4.2.2.- Obtención de información

Si no ha habido errores durante los primeros intercambios de mensajes, la siguiente acción recae sobre el controlador. En este paso el controlador averigua las capacidades de cada uno de los dispositivos de red. Esto se obtiene mediante un mensaje de tipo **Feature request** (Petición de características) que es enviado por el controlador de la red a los distintos elementos de la red. Éstos envían una respuesta que contiene todas sus capacidades. Este mensaje de respuesta es inmediatamente precedido de otro del tipo **Get Configuration Request** (Petición de configuración) en donde los switches le envían al controlador su estado de configuración actual. De manera adicional, el controlador puede enviar otras peticiones a dispositivos de la red con mensajes como **Port status**, para conocer el estado de sus puertos. En la figura 2.4 se muestra un diagrama con los distintos mensajes que se envían entre controlador y switches de una red en su fase de iniciación y mantenimiento de la conexión.

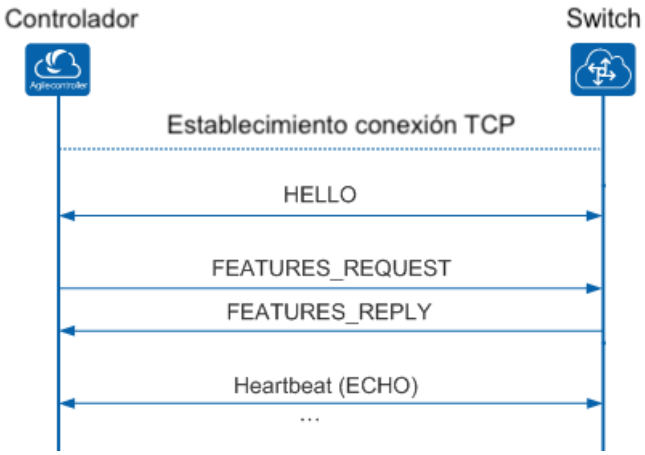


Figura 2.4.- Establecimiento y mantenimiento de la conexión con el canal OpenFlow [20].

2.4.2.3.- Análisis de rutas y topología de red

Una vez establecida esta conexión entre controlador y switch, el protocolo OpenFlow especifica que el controlador debe «descubrir» la ubicación de los switches de la red. De lo que en realidad se trata este descubrimiento es la generación por parte del controlador de la topología de la red. Esto quiere decir, que el controlador utiliza

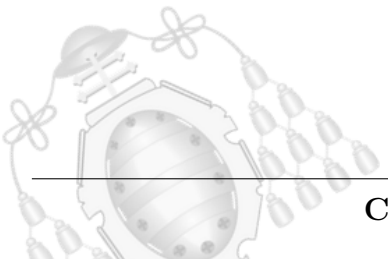
una serie de mecanismos a través de los cuales obtiene la suficiente información como para reconstruir un mapa o modelo de la red que debe de gestionar. Estos mecanismos de averiguación de la topología de la red se realizan mediante un protocolo de descubrimiento, denominado **LLDP** (Link Layer Discovery Protocol).

A través de este proceso el controlador es capaz de recrear con exactitud la topología de la red a gobernar. Para ilustrarlo de una manera mucho más práctica, se adjunta un ejemplo explicativo en el Anexo VI.

2.4.3.- Versiones de OpenFlow

Desde el lanzamiento de la primera versión estable, la 1.0.0 [8], han sido numerosos los cambios producidos [23] en las distintas versiones del protocolo.

Dos años después de la versión 1.0.0, la Open Networking Foundation sacaba a la luz la primera gran revisión, la 1.1.0 [9]. Entre otras novedades, destacaba el soporte para las denominadas «tablas de grupo». Su principal objetivo es agrupar acciones u operaciones comunes pertenecientes a un conjunto de flujos de tráfico. Asimismo, también permite la posibilidad de utilizar varias «tablas flow», aumentando la capacidad de mapeo y redireccionamiento de tráfico. Se incluye también el soporte para etiquetado en redes LAN virtuales (*VLAN tagging*). En la figura 2.5 se muestra un pequeño diagrama del proceso de *pipeline* diseñado en la versión 1.1.0 de OpenFlow.



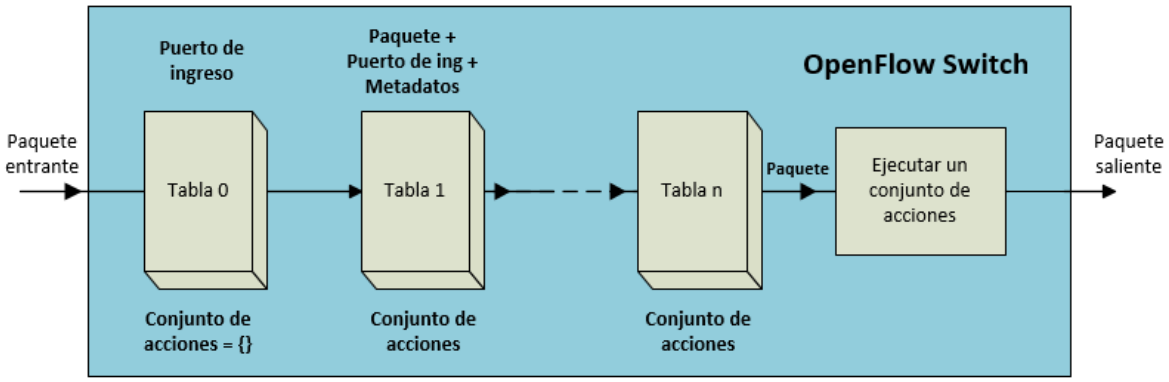


Figura 2.5.- Esquema del proceso de «entubado» o *pipeline* diseñado en la versión 1.1.0 del protocolo OpenFlow [9].

La versión 1.2.0 [10] añade soporte para direccionamiento IPv6. Define 44 campos [30] por defecto, entre los que se incluye, por ejemplo, la separación de puertos TCP y UDP y amplía la capacidad de modificación de los paquetes. Esta versión sienta las bases para una alta disponibilidad del controlador, permitiendo la conexión de un mismo dispositivo a varios controladores. Se definen tres roles para el controlador:

- **Equal**: Con capacidad total para modificar un switch. Cada switch puede tener N controladores equal.
- **Master**: También con capacidad total para modificar un switch. Cada uno puede tener únicamente asociado un controlador máster.
- **Slave**. Con capacidad únicamente de lectura, cada switch puede tener asociados N controladores slave.

La especificación de la versión 1.3.0 [11] abre la posibilidad de incorporar medidores de flujos de tráfico. Esto permite regular los flujos de tráfico con el objetivo de adaptarlos para ofrecer diferentes calidades de servicio en función de las aplicaciones que se estén utilizando (Quality of Service, QoS). También cambia el comportamiento del conmutador o switch en caso de que un paquete no coincida con ninguna de las

reglas de sus tablas «Flow»: En vez de etiquetarse con un «flag», lo que se ejecuta es una regla específica, a definir por el administrador de la red.

La versión 1.4.0 [21] se centra sobre todo en la corrección de errores. Añade la característica de «sincronización entre tablas» y cambia el puerto TCP por defecto del 6633 al 6653.

En su versión más reciente, la 1.5.0, [12] se introducen nuevas mejoras así como la posibilidad de monitorizar flujos de tráfico. En la figura 2.6 se muestra un esquema general de los principales elementos que conforman un switch OpenFlow tal y como establece esta especificación.

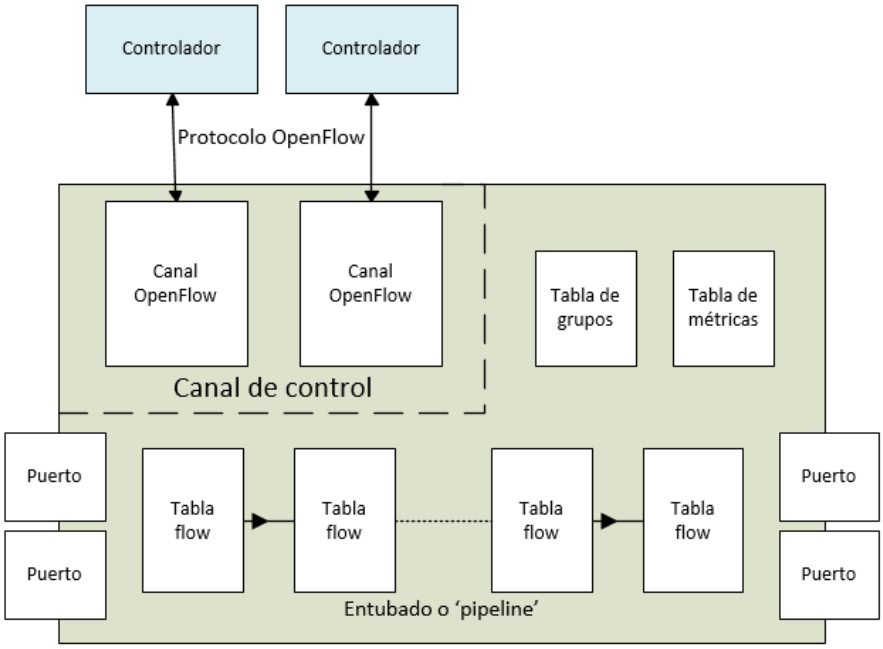


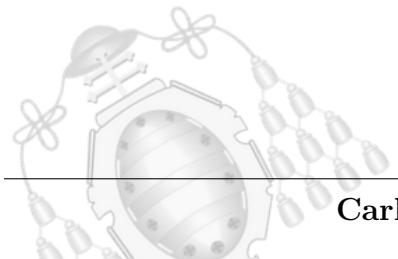
Figura 2.6.- Esquema de los principales componentes que conforman un switch OpenFlow según la especificación 1.5.0 [12].

2.5.- Seguridad perimetral

A medida que el equipamiento de las redes ha ido sumando nuevas y sofisticadas funcionalidades, de forma paralela se han introducido mejoras en la gestión de su seguridad: nuevos protocolos de enrutamiento, políticas de acceso o división lógica de las redes son algunos ejemplos.

La centralización de la lógica de la red supone ciertas ventajas en cuanto su configuración, versatilidad y rapidez a la hora de implantar medidas y políticas de seguridad. En una red tradicional, las acciones llevadas a cabo por el administrador para preservar su seguridad recaen, en buena parte, en una correcta configuración de los equipamientos: creación y división de la red en segmentos lógicos, configuración de protocolos y puertos en los equipos que dan soporte a la red, establecimiento de listas de control de acceso, etcétera.

Resulta de interés aprovechar las ventajas que ofrecen las SDN e investigar en el desarrollo de aplicaciones que preserven la seguridad en la red frente a posibles ataques comunes: Denegación de servicio, envío de paquetes fraudulentos, envenenamiento de rutas o captación de información son algunos de los ejemplos más comunes. El ámbito de las redes SDN se encuentra todavía en fase de evolución y explotación, máxime si se trata del caso particular de las aplicaciones. Si bien su número aumenta periódicamente, pocas profundizan en temas de seguridad más allá de detección de ataques por denegación de servicio o aseguramiento de la conectividad segura a nivel de protocolo OpenFlow.



3. Desarrollo

En este capítulo se detalla el entorno de trabajo objeto de esta investigación. También se mencionan las distintas herramientas y programas utilizados.

3.1.- Entorno de trabajo

El entorno de trabajo es eminentemente virtual, al no disponer de un equipamiento físico adecuado para ello. Sin embargo, esto no resta interés al objeto de esta investigación, puesto que las aplicaciones son independientes de la infraestructura de red. Esto quiere decir que, si en un futuro se contase con un equipamiento de red real, las aplicaciones desarrolladas en este proyecto deberían funcionar sin ningún inconveniente, siempre y cuando los protocolos sean los mismos, en este caso, OpenFlow.

A la hora de diseñar el entorno, es conveniente seguir una serie de directrices y pautas con el fin de obtener aquellos recursos necesarios que más se adecuen para el fin de esta investigación. Estas pautas son las siguientes:

- **Elección de los elementos de red**: Todo el equipamiento que forme parte de la red debe cumplir con una serie de requisitos, entre ellos, su compatibilidad con el estándar OpenFlow.
- **Elección del entorno de simulación**: Una vez se haya o hayan escogido correctamente estos elementos, se debe encontrar el software sobre el que se va a realizar la simulación la red. Se tiene en cuenta, para ello, el equipamiento físico con el que se cuenta para este proyecto. También que dicho entorno permita la inclusión de máquinas externas con sus correspondientes sistemas operativos.
- **Elección del controlador**: Por último, decantarse por un controlador de red, en base a criterios de facilidad de desarrollo de aplicaciones, así como soporte y documentación.

Durante los siguientes apartados, se describen cada una de las alternativas a estudiar para la implementación del entorno de trabajo, así como las soluciones finalmente adoptadas para el presente proyecto.

3.1.1.- Virtualización

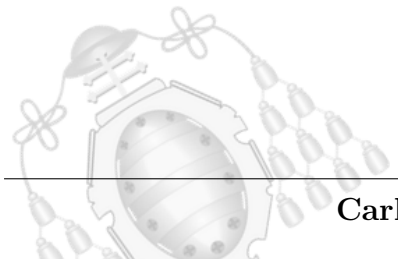
Durante el transcurso de este trabajo, se hace mención de forma recurrente a este término. Es por ello que conviene explicar en qué consiste y los tipos de virtualización que existen a día de hoy.

La virtualización es una tecnología desarrollada por IBM en los años 60, aunque no fue hasta el final de los 90 cuando verdaderamente irrumpió, no solo en el área tradicional de los servidores, sino también en muchos otros ámbitos del mundo de la computación.

Cuando se habla de virtualización, se hace alusión a la abstracción de los recursos de una computadora, comúnmente denominada «hipervisor» o Monitor de Máquina Virtual (VMM), entre el hardware de la máquina física (o host) y el sistema operativo de la máquina virtual (en inglés, *guest*). Es el Hipervisor el encargado de manejar, gestionar y arbitrar los cuatro recursos principales de una computadora: CPU, Memoria, Red y Almacenamiento, con el fin de repartir de una manera dinámica tales recursos entre todas las máquinas virtuales del host.

3.1.1.1.- Tipos de virtualización

Se habla de virtualización «nativa», *bare metal* o de tipo 1 cuando no existe ningún sistema operativo de por medio entre el hardware y el hipervisor. Un ejemplo de ello es VMWare ESXi, puesto que se instala como un sistema operativo sobre una máquina o servidor y es sobre el que se ejecutan todas y cada una de las máquinas virtuales. Es la mejor solución para grandes servidores, puesto que son más estables y permiten aprovechar mucho mejor los recursos de la máquina.



Otra virtualización es la «no nativa», *hosted* o de tipo 2. Es la más extendida a nivel de usuario en donde entre el hipervisor y el hardware hay un sistema operativo (Linux, Windows, MacOS). Un ejemplo es VirtualBox, programa que se instala sobre un sistema operativo de los anteriormente mencionados. Son más sencillos de implementar a pesar de que se aprovechan en menor medida los recursos de la computadora. En la figura 3.1 se muestra un esquema comparativo entre ambos tipos de virtualización.

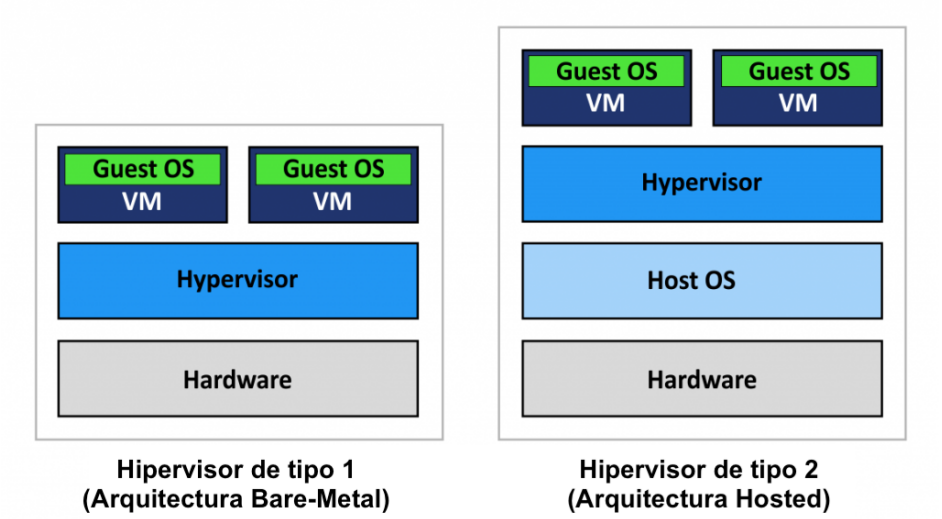


Figura 3.1.- Esquema diferencia entre virtualización de tipo 1 y de tipo 2 [36].

3.1.2.- Elección de los elementos de red

Al no poder contar con equipamiento real, resulta indispensable poder elegir adecuadamente qué elementos virtuales pueden reemplazarlos. Esta elección se basa, fundamentalmente, en dos criterios:

- **Soporte de OpenFlow:** Es el principal requisito puesto que, como se ha comentado en anteriores apartados (véase apartado 2.4), es la base de las redes definidas por software. Si bien existen múltiples protocolos propietarios, OpenFlow se erige como el estándar de facto en las SDN.

  Tangencial a este requisito, está el de las versiones de OpenFlow soportadas por dicho dispositivo. Entre las múltiples existentes, una de las más extendidas y que cubre perfectamente las necesidades de este proyecto es la versión 1.3.

- **Consumo de recursos:** Dadas las limitaciones del equipo físico con el que se cuenta, es un aspecto muy a tener en cuenta. Es conveniente que sea un dispositivo virtualizado con un bajo consumo de recursos, tanto de memoria como de procesador y que ello no redunde en un bajo rendimiento. Hay que ser conscientes de que el aprovisionamiento de recursos en el entorno de trabajo debe repartirse, no solamente en estos equipos de la red, sino también en las máquinas que simulan los ataques y aquéllas que hagan de usuarios destinatarios de los mismos, así como del controlador de red.

3.1.2.1.- OpenSwitch

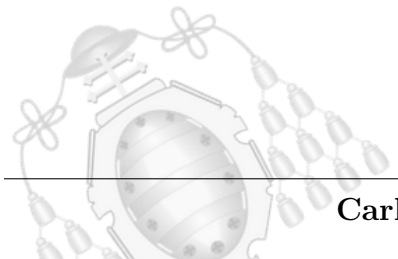
Es un proyecto iniciado por HP basado en un switch virtual sobre Linux. Adicionalmente, se quiere con ello impulsar una comunidad de desarrolladores que prioricen el uso de sistemas abiertos y evitar la dependencia de otros propietarios, así como aumentar la interoperabilidad.

Cuenta con la participación de Intel y VMWare entre otros, junto al anteriormente mencionado HP, y cuenta con amplia compatibilidad con sistemas abiertos como OpenFlow.

3.1.2.2.- Open vSwitch

Open vSwitch - abreviado, OVS - es una plataforma de código abierto (*open source*) que consiste en un switch multicapa virtual, con el objetivo de proveer soporte a redes fundamentalmente virtualizadas. Nació en el año 2009 bajo la licencia 2.0 de Apache y hoy en día es utilizado por multitud de plataformas a nivel internacional, así como los principales software de virtualización, tales como VirtualBox o VMWare.

Cuenta con integración con diversos protocolos, como por ejemplo NetFlow, sFlow o LACP. Soporta también la creación de redes LAN virtuales (VLAN) y desde hace varios años se integra junto a OpenFlow en todas sus versiones. [47]



El proyecto estaba dirigido principalmente para ejecutarse en entornos virtualizados sobre sistemas operativos basados en Linux, aunque ha ido extendiendo sus límites hasta incluso equipamientos físicos. También es especial su relevancia en entornos virtualizados compuestos por más de un servidor físico, como muestra la figura 3.2.

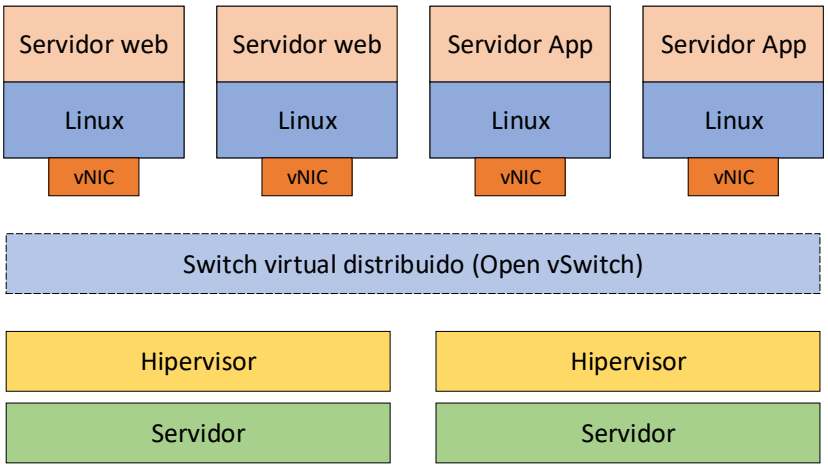


Figura 3.2.- Open vSwitch como una infraestructura totalmente transparente en un entorno virtualizado formado por dos servidores físicos [49].

Uno de los principales objetivos de OVS es proveer un entorno de configuración con una sintaxis sencilla y amplia compatibilidad entre todas las distribuciones de Linux hasta tal punto que, en muchas, se encuentra instalado por defecto.

Dada su facilidad de uso, su interoperabilidad así como contar con una amplia comunidad y soporte, se convierte en un firme candidato para este proyecto. En la tabla 3.1 se adjuntan los requerimientos mínimos en un sistema de OVS.

Requerimiento	**Valor mínimo**	**¿Cumple?**
Sistema Operativo	MacOS, Windows, Linux	Sí
Memoria (RAM)	512MB	Sí
Almacenamiento	200MB	Sí
Python	v3.4 o superior	Sí
CPU	x86, 64 bit	Sí

Tabla 3.1.- Requisitos mínimos de Open vSwitch [47].

3.1.3.- Elección del entorno de simulación

Una vez determinados los dispositivos de red a usar, el siguiente paso es buscar el mejor entorno posible para su implementación. La paleta de soluciones se reduce a dos grandes opciones: la utilización de un software para la virtualizacion de sistemas operativos, o apostar por programas orientados a la emulación de redes.

No son pocas las alternativas de las que se disponen hoy en día en el mercado y su elección depende, en buena medida, del equipamiento con el que se cuente para dicha simulación y sus características, tales como el tipo de procesador, su arquitectura o la cantidad de memoria, entre otras.

Otro requisito importante a tener en cuenta es la posibilidad que ofrezca de poder introducir en el escenario de red máquinas reales, con sistemas operativos propios. Esto es importante, puesto que todos los ataques objeto de estudio requieren de herramientas cuyo soporte o sistema operativo pertenece bien a Windows, MacOS o Linux.

Para cada una de ellas, se realiza un estudio pormenorizado de los requerimientos mínimos para su ejecución, su compatibilidad con la estación de trabajo que los aloje y si es posible o no crear interfaces de red virtuales con ellos, con el fin de interconectar las máquinas virtuales y el equipamiento.

Se parte de una estación de trabajo que cuenta con las especificaciones que se muestran, reflejadas en la tabla 3.2

Elemento	**Valor**	**Tipo**
Sistema Operativo	MacOS Catalina 10.15.4	64bits
Memoria (RAM)	2x4GB	DDR3@1600MHz
Almacenamiento (1)	500GB	SSD
Almacenamiento (2)	1TB	HDD
CPU	Intel®Core™i5	4 núcleos @2.7GHz

Tabla 3.2.- Especificaciones de la máquina a utilizar en el entorno de trabajo.

3.1.3.1.- Virtualización nativa o no

Para proyectos de esta embergadura, y al no disponer de equipamiento físico, es deseable apostar por una virtualización nativa o de tipo 1, ejecutándose en equipamiento dedicado para tal fin, como racks o módulos servidores.

Dadas las circunstancias de crisis sanitaria, durante las cuales cursa este proyecto, no es posible disponer de dicho equipamiento. Es por ello que se utiliza una virtualización de Tipo 2 o *hosted*.

3.1.3.1.1.- VMWare workstation

Es un programa desarrollado por VMware Inc [37]. Cuenta con más de 20 años de desarrollo y es uno de los software de virtualización más extendidos en entornos profesionales.

VMWare se erige, no solamente como un producto único, sino como un ecosistema de herramientas y aplicaciones conectadas: orientadas desde el almacenamiento de datos de forma distribuida hasta aplicaciones de seguridad pasando por el almacenamiento en redes Cloud.

Tiene un período de evaluación de 30 días, con un coste de 166€ para un período de 3 años. Este punto penaliza su utilización frente a otras alternativas existentes en el mercado, como VirtualBox que son completamente gratuitas. En la tabla 3.3 se muestran los requerimientos mínimos de VMWare.

Requerimiento	**Valor mínimo**	**¿Cumple?**
Sistema Operativo	MacOS, Windows, Linux	Sí
Memoria (RAM)	2GB	Sí
Almacenamiento	5GB	Sí
CPU	CPU 2011 o superiores, velocidad>1.3GHz	Sí

Tabla 3.3.- Requisitos mínimos de VMWare [37].

3.1.3.1.2.- VirtualBox

Se trata de un software de virtualización multiplataforma [38], disponible tanto para Linux, Windows como MacOS. Desarrollado por Oracle Corporation desde el año 2007, soporta la virtualización de multitud de sistemas operativos, desde Windows hasta MacOS, así como la mayoría de distribuciones de Linux.

Una de sus grandes bazas es la facilidad de manejo por parte de casi cualquier tipo de usuario. Entre sus características más destacadas, están la ejecución de máquinas de manera remota por medio del protocolo *Remote Desktop Protocolo* (RDP), soporte de almacenamiento de discos utilizando iSCSI o la aceleración en 3D gracias a un paquete de controladores desarrollados por Oracle.

A pesar de contar con características más limitadas frente a su homólogo de VMWare, su licencia es gratuita y reúne todos los requisitos para su utilización en este trabajo de investigación. En la tabla 3.4 se adjuntan los requerimientos mínimos para la ejecución de VirtualBox.

Requerimiento	**Valor mínimo**	**¿Cumple?**
Sistema Operativo	MacOS, Windows, Linux	Sí
Memoria (RAM)	512MB	Sí
Almacenamiento	30MB	Sí
CPU	Soporte virtualización x86	Sí

Tabla 3.4.- Requisitos mínimos de VirtualBox [38].

3.1.3.2.- Simulador de redes

Además de utilizar software para la virtualización de sistemas operativos, otra posibilidad es el empleo de programas específicos para la simulación de redes de comunicaciones. Existen en el mercado multitud de opciones, tanto propietarias como de código abierto.

3.1.3.2.1.- Packet Tracer

Packet Tracer es un software de licencia propietaria de la compañía Cisco Systems que permite la creación y simulación de topologías de redes basadas en equipamiento de Cisco, como switches o routers.

Provee a cada equipamiento simulado de una interfaz por línea de comandos, emulando a las que utiliza en el sistema operativo de sus dispositivos, Cisco iOS.

Si bien el equipamiento de Cisco permite su integración con el protocolo OpenFlow, este programa de simulación no permite la conexión del equipamiento simulado con un controlador o máquina externa al programa. Por tanto, este simulador de redes no se adecúa con los requerimientos de este trabajo. Además, tampoco permite incluir máquinas externas con sistema operativo propio.

3.1.3.2.2.- NS-3

Es un emulador de redes libre basado en eventos discretos. Permite emular todo tipo de redes, tanto inalámbricas como cableadas, aunque su uso se centra fundamentalmente en redes inalámbricas ad-hoc.

Programado en C++ y con soporte en Python desde su versión 3, es uno de los programas más extendidos en el ámbito educativo. Cuenta con multitud de librerías y su uso libre lo convierten en una herramienta muy potente.

Existen proyectos de SDN realizados con NS-3, aunque estos se limitan a ámbitos muy concretos en el entorno inalámbrico y no permite la inclusión de elementos externos, como por ejemplo el uso de máquinas virtuales en sus redes.

3.1.3.2.3.- GNS3

GNS3 es un emulador gráfico multiplataforma muy potente que permite el diseño de topologías de red muy complejas. Es un programa gratuito y de uso bastante

sencillo para un usuario medio. Una de las múltiples características que incorpora es la posibilidad de utilizar imágenes binarias de equipamiento de redes reales, como por ejemplo, los sistemas de Cisco iOS. Ello permite una simulación de red muy fiel a la realidad en un entorno virtual.

Otra de las características que brinda este programa es la posibilidad de integrar máquinas virtuales en el entorno de la red, simulando ser usuarios. Esto lo diferencia de otros programas de simulación, en donde las máquinas finales eran simples consolas de comandos con unas funciones muy limitadas.

La versatilidad de este programa, y su integración con casi cualquier tipo de máquinas virtualizadas, lo convierten en un claro candidato para su utilización en el entorno de trabajo.

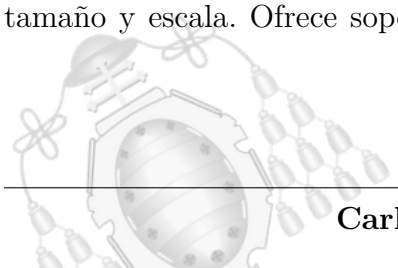
3.1.4.- Controladores de red

El número de controladores de red para este tipo de redes ha ido creciendo de manera casi exponencial desde la aparición de NOX, el primer controlador de redes SDN. Con el transcurso del tiempo, la aparición de nuevos controladores ha ido ligada también a una ampliación de características de los mismos. Entre los más populares a día de hoy destacan OpenDayLight, POX -versión de NOX escrita en Python-, Floodlight y Ryu. De todos ellos se analizan, entre otras características, su integración con el protocolo OpenFlow y la facilidad de diseño de aplicaciones que ofrecen. También su soporte y consumo de recursos.

De este análisis se obtiene el controlador que más se ajusta a los propósitos iniciales del presente Trabajo Fin de Máster

3.1.4.1.- OpenDayLight

Se trata de un software modular para personalizar y manejar redes de cualquier tamaño y escala. Ofrece soporte para multitud de protocolos, entre ellos OpenFlow,



NETCONF o SNMP. Está escrito en Java y su proceso se ejecuta en un entorno JVM (Java Virtual Machine).

Permite al usuario, mediante el uso de una interfaz por línea de comandos, configurar, instalar y eliminar todo tipo de dependencias en el controlador, como pueden ser aplicaciones o plugins.

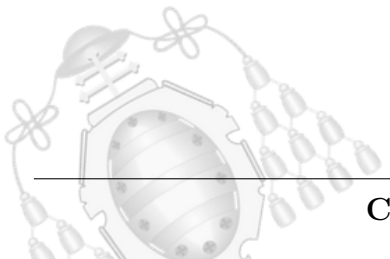
La versión más reciente de OpenFlow soportada por OpenDayLight es la 1.3. Existen diversos proyectos paralelos para dotar a este controlador de soporte para las versiones posteriores, sin embargo, estos proyectos no están secundados por los principales desarrolladores del controlador.

En las últimas versiones (Magnesium y Sodium) se han eliminado numerosas dependencias que facilitaban la interacción con el controlador por medio de interfaces gráficas.

El desarrollo de aplicaciones se realiza en lenguaje Java. Cuenta en la actualidad con multitud de módulos de desarrollo de aplicaciones, así como APIs REST para utilizar en la comunicación tanto Northbound como Southbound. En la tabla 3.5 se muestran los requerimientos mínimos para la ejecución de OpenDayLight.

Requerimiento	**Valor**	**¿Cumple?**
Versión O.F más reciente	1.3	Sí
Lenguaje desarrollo	Java	Sí
RAM mínima	2GB	Sí
CPU mínima	2 núcleos	Sí
Almacenamiento mínimo	16 GB	Sí
Soporte de S.O	Linux con JVM > 1.7	Sí

Tabla 3.5.- Requisitos mínimos de OpenDayLight [35].



3.1.4.2.- ONOS

Es un controlador Open Source perteneciente a la Open Network Foundation (ONF). Provee todo tipo de herramientas y módulos, entre ellos de aplicaciones, para el manejo de redes definidas por software.

Cuenta con una serie de APIs REST para su control por medio de diversos programas y, al igual que OpenDayLight, está desarrollado enteramente en Java, dotándolo de una mayor interoperabilidad.

ONOS [39] permite, entre otras muchas características, su ejecución en un entorno distribuido, aprovechando los recursos de CPU y memoria de múltiples servidores, dotándolo de una mayor tolerancia ante fallos.

Desde el mes de julio de 2016, cuenta con soporte para la versión 1.5 de OpenFlow, la más reciente hasta el momento. En la tabla 3.6 se muestran los requisitos mínimos para la ejecución de ONOS.

Requerimiento	**Valor**	**¿Cumple?**
Versión O.F más reciente	1.5	Sí
Lenguaje desarrollo	Java	Sí
RAM mínima	2GB	Sí
CPU mínima	2 núcleos	Sí
Almacenamiento mínimo	10 GB	Sí
Soporte de S.O	Linux con JVM $>$ 1.8	Sí

Tabla 3.6.- Requisitos mínimos de ONOS [39].

3.1.4.3.- POX

POX [40] es la versión con soporte en lenguaje Python de NOX, el primer controlador de la breve historia de las SDN. Fue inicialmente desarrollado por Nicira Networks en el año 2009, si bien ahora el proyecto está a cargo de VMWare.

Tiene a cargo una extensa documentación [41] entre la que se encuentra el desarrollo de aplicaciones en su entorno de programación. POX viene incluido en programas como Mininet [42], un emulador de redes SDN basado también en Python.

Existe un proyecto, adicional a POX, denominado POX Web GUI [43] que permite de una forma sencilla monitorizar toda la red desde el controlador: desde consultar la tabla de flujos de los dispositivos de red hasta mirar qué tipos de paquetes han sido recibidos en cada uno de sus puertos.

Las aplicaciones están escritas en Python, gracias al desarrollo de un módulo por parte de la comunidad de POX. Permite su uso por medio del intérprete de comandos estándar (CPython) y también por medio de PyPy [44].

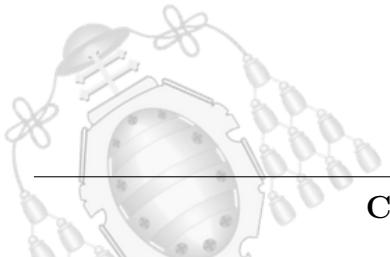
La facilidad de su lenguaje de programación, así como los requerimientos mínimos para su ejecución lo convierten en un claro candidato para este proyecto. En la tabla 3.7 se muestran los requisitos mínimos para la ejecución de POX

Requerimiento	**Valor**	**¿Cumple?**
Versión O.F más reciente	1.5	Sí
Lenguaje desarrollo	Python	Sí
RAM mínima	512MB	Sí
CPU mínima	2 núcleos	Sí
Almacenamiento mínimo	1 GB	Sí
Soporte de S.O	Linux, MacOS, Windows	Sí

Tabla 3.7.- Requisitos mínimos de POX [41].

3.1.4.4.- Ryu

Siguiendo la estela de POX, Ryu se ofrece como otra alternativa de controlador basado en Python. Su filosofía [45] se centra en dos aspectos bastante claros:



- **Agilidad** proporcionando un Framework para el desarrollo de aplicaciones muy potente y sencillo de utilizar para casi cualquier tipo de usuario con conocimientos básicos de programación.
- **Flexibilidad** gracias a un conjunto de APIs para la comunicación entre controlador y aplicaciones.

El desarrollo de aplicaciones sigue una metodología modular. Cada aplicación consiste en uno o varios componentes proporcionados por Ryu, tales como switches, firewalls o sistemas de detección de intrusiones (IDS). Todos ellos son perfectamente modificables y combinables por parte del usuario, con el fin de construir sus propias aplicaciones.

Altamente recomendada por algunos fabricantes de dispositivos y especialmente diseñado para integrarse con Open vSwitch. Al igual que POX, la facilidad de su lenguaje de programación, los requerimientos mínimos para su ejecución y su alta compatibilidad con Open vSwitch lo convierten en un firme candidato para este proyecto. En la tabla 3.8 se muestran los requisitos mínimos para la ejecución de Ryu.

Requerimiento	**Valor**	**¿Cumple?**
Versión O.F más reciente	1.5	Sí
Lenguaje desarrollo	Python	Sí
RAM mínima	512MB	Sí
CPU mínima	2 núcleos	Sí
Almacenamiento mínimo	512 MB	Sí
Soporte de S.O	Linux y todas sus distribuciones	Sí

Tabla 3.8.- Requisitos mínimos de Ryu [34].

3.2.- Herramientas y programas

De manera adicional, se cuenta en este proyecto con una serie de aplicaciones enfocadas tanto a la captura como generación de tráfico en la red, así como diversas herramientas para el escaneo de servicios.

3.2.1.- Scapy: Manipulación avanzada e interactiva de paquetes

Scapy es una herramienta de generación, manipulación y captación de paquetes en la red muy potente, con soporte para la mayoría de protocolos existentes en el ámbito de Internet. Su versatilidad y total libertad para el usuario a la hora de capturar, modificar o crear paquetes hacen de este programa su punto fuerte.

Escrito en lenguaje Python y con soporte para todos los sistemas Linux, cuenta con una versión tanto gráfica como por consola. También es posible hacer uso de Scapy por medio de scripts en lenguaje Python, siempre y cuando se incluya al principio del archivo las sentencias de importación requeridas.

Para este proyecto, se hace uso de esta última opción y de Scapy-OpenFlow, una versión de Scapy adaptada al protocolo OpenFlow y que incluye, entre otras características, librerías y funciones de generación de paquetes de este protocolo utilizado en las redes definidas por software.

3.2.2.- Wireshark: Captura y análisis de tráfico en la red

Se trata de una herramienta ampliamente extendida y su utilización principal se centra en el análisis y solución de problemas en redes de comunicaciones. Permite el análisis y captura de tráfico de multitud de protocolos y cuenta con versión tanto gráfica como en modo texto, esta última denominada *tshark*. Su funcionamiento es muy similar a la de TCPDump, otra herramienta multiplataforma de captura de paquetes.

Es multiplataforma, pudiendo ser ejecutada en sistemas operativos tipo Unix (incluyendo Linux, Solaris o MacOS) o Microsoft Windows.

3.2.3.- nmap: Escaneo en la red

Se trata de una utilidad gratuita y de código abierto, usada principalmente para el descubrimiento de redes y auditorías de seguridad. Mediante el uso de técnicas novedosas [33], nmap es capaz de determinar, entre otros datos, el número de

dispositivos activos en la red, su sistema operativo y versión, los distintos servicios que ofrece o si utilizan cortafuegos.

Ofrece al usuario multitud de posibilidades para realizar escaneos dentro de una red: Por rango de puertos, por un puerto específico, protocolos específicos o varias subredes a la vez entre otras opciones.

Está especialmente diseñada para un descubrimiento rápido en redes grandes y complejas. Cuenta también con una versión gráfica (Zenmap). Es multiplataforma, estando disponible tanto para Linux, MacOS como Windows.

3.3.- Decisión de elementos

En las secciones previas se ha expuesto diversas opciones de cara a construir el escenario de simulación. Se parte de un entorno simulado que contiene máquinas virtuales que emulan el comportamiento tanto del equipamiento de los usuarios finales como el de red.

- Como software de emulación de los dispositivos de la red se utiliza **Open vSwitch**. Éste se aloja en una máquina virtual con el sistema operativo Ubuntu Server 20.12 LTS, 1GB de memoria RAM, 10GB de almacenamiento.
- Para el controlador de red se utiliza **Ryu**, también alojado en una máquina virtual Ubuntu Server 20.12 LTS, con 1GB de memoria RAM, 10GB de almacenamiento. La elección de Ryu, en detrimento de POX se hace en base a la documentación aportada en su página web, así como contar con multitud de ejemplos y un mayor soporte en la comunidad de desarrolladores. Así mismo, cuenta con mayor número de actualizaciones que POX.
- Para simular todo el escenario, se va a hacer uso de un emulador de redes, en concreto **GNS3**, dada la gran optimización que GNS3 cuenta para sistemas operativos MacOS. Otra posibilidad era la utilización de VirtualBox e interconectar los equipos por medio de interfaces virtuales que proporciona el programa. Sin embargo, este segundo enfoque es mucho más rígido en cuanto a configuración, sobre todo si los escenarios a simular cambian bastante.

3.4.- Construcción del escenario

Como se ha explicado en el apartado 3.1.3.2.3, GNS3 es un potente simulador gráfico de redes y permite introducir en el escenario tanto máquinas «simuladas» como máquinas virtuales «reales».

Las máquinas «simuladas», denominadas VPCS (virtual PCs) consisten en pequeños terminales CMD que permiten realizar pruebas de conectividad.

En cuanto a los elementos de red, es posible añadir tanto switches como routers. Por defecto, vienen los dispositivos mostrados en la figura 3.3.

Figura 3.3.- Lista de dispositivos disponibles por defecto en GNS3.

Para este proyecto, se parte de un escenario simple, con un switch, el controlador, la máquina atacante y la víctima, como se refleja en la figura 3.4

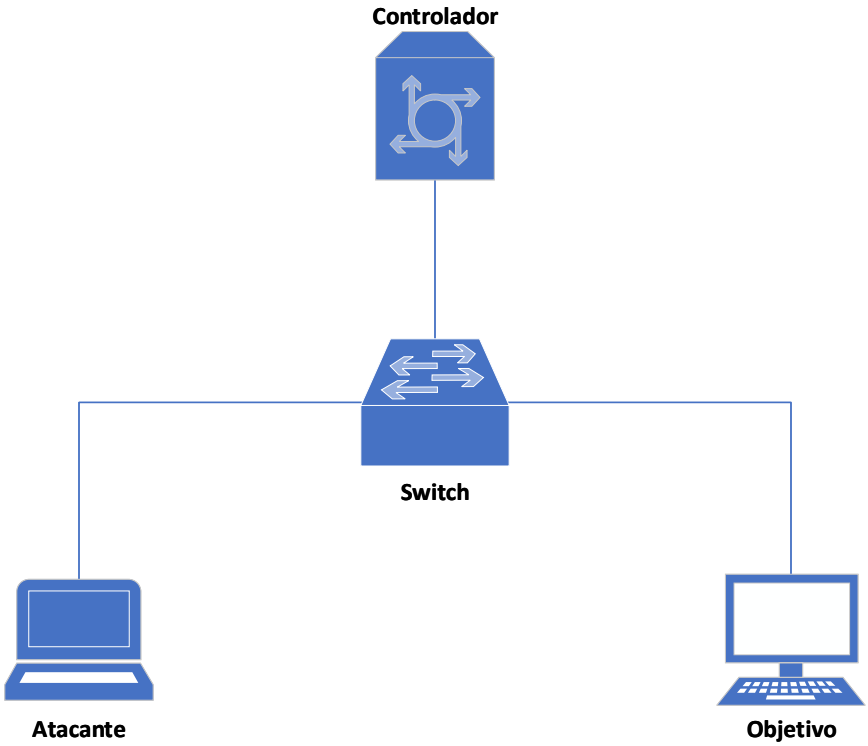


Figura 3.4.- Maqueta del primer escenario a realizar en GNS3.

Todo ello está realizado con los elementos que vienen por defecto en el programa GNS3. Sin embargo, esto no sirve para los propósitos de este proyecto y es necesario adecuarlo convenientemente:

- El switch debe ser Open vSwitch. Esto requiere la realización una serie de pasos para su inclusión en el presente proyecto, que se comentan más adelante.
- El controlador es una máquina virtual Ubuntu Server, con las características especificadas en el apartado 3.3
- El atacante se sustituye por una máquina virtual con el sistema operativo Kali Linux, con 2GB de RAM y 20GB de almacenamiento.
- El objetivo del ataque es una máquina virtual Ubuntu Server, con las mismas características que el controlador, especificadas en el apartado 3.3.

3.4.1.- Integración de OVS

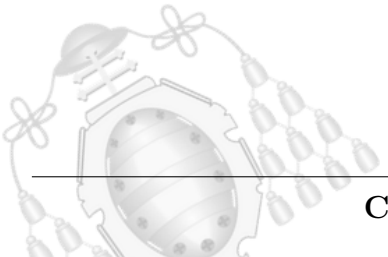
La propia página de GNS3 ofrece, en su apartado de descargas, una plantilla [48] para generar un switch OVS y poder integrarlo en el escenario. Cada uno dispone de 16 puertos distintos, existiendo dos variantes:

- Con interfaz de gestión: Cuenta con un puerto dedicado exclusivamente para ser conectado al controlador.
- Sin interfaz de gestión: A diferencia del anterior, cualquier puerto puede ser utilizado para conectarse al controlador.

Se hace uso del switch con una interfaz de gestión específica, puesto que permite realizar de una manera paralela una red de gestión, separada del plano de datos, aislando al controlador del resto de usuarios de la red.

Los pasos para integrarlo en el escenario son bastante sencillos. En la página de GNS3 se remite a una dirección para descargar un archivo con extensión **gns3**. Dicho fichero realiza la descarga de un entorno virtualizado por medio de la tecnología Docker, instalando las dependencias necesarias para ejecutar Open vSwitch. Esto reduce enormemente el consumo de recursos con respecto a su utilización en una máquina virtual convencional.

Una vez realizados todos los pasos, aparece el nuevo switch disponible en la paleta de dispositivos y se permite su importación al escenario. El resultado final se muestra en la figura 3.5, donde puede verse un escenario compuesto por un switch Open vSwitch y tres elementos conectados a él: el controlador, el atacante y el objetivo del ataque.



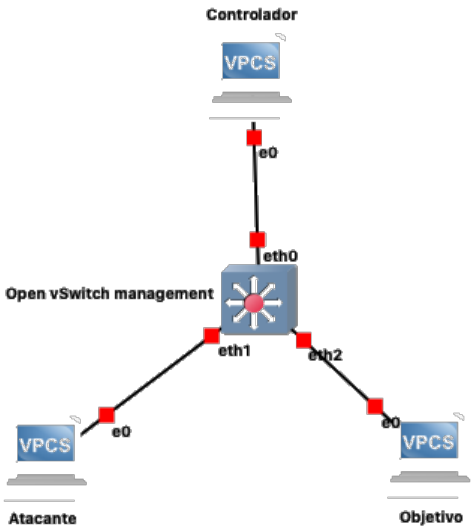


Figura 3.5.- Primer escenario, con el switch OVS introducido.

La elección del puerto eth0 en el switch para conectar el controlador no responde a criterios arbitrarios. Tal y como establece la documentación de Open vSwitch para GNS3, este puerto está reservado para tal fin. El resto pueden utilizarse indistintamente para cualquier otro propósito: conexión de otros equipamientos, estaciones de usuario, máquinas virtuales o, llegado el caso, un segundo o varios controladores de respaldo.

El siguiente paso es la inclusión de las máquinas virtuales correspondientes a la máquina atacante y al controlador de red.

3.4.2.- Importación de las máquinas virtuales

GNS3 permite el uso dentro de su simulador de máquinas virtuales ubicadas en el mismo PC. Este aspecto resulta especialmente útil en el ámbito del pentesting o auditoría de redes, máxime si el equipamiento de red simulado utiliza el mismo sistema operativo que sus homólogos en hardware, como puede ser el caso de los firmware Cisco iOS.

En el ámbito de este proyecto de investigación resulta vital en dos aspectos:

- Posibilita el uso de un controlador de red software.

- Permite utilizar máquinas específicas para auditorías de seguridad y ejecutar diversos ataques controlados, como es el caso de la distribución Kali Linux.

La virtualización de estas máquinas está ligada a un software externo, bien VirtualBox o VMWare. GNS3 permite sincronizarse con estos programas y utilizar todas las máquinas que ellos dispongan. Se aprovecha el anterior entorno realizado con VirtualBox y se hace uso de las mismas máquinas. La máquina objetivo sigue siendo un PC virtual del propio programa. El escenario resultante se muestra en la figura 3.6

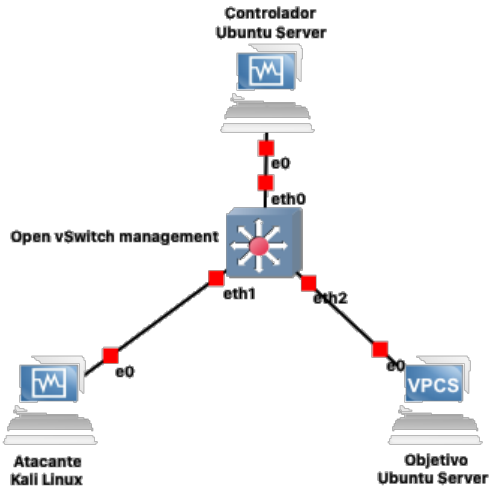


Figura 3.6.- Primer escenario, con el switch OVS y las máquinas virtuales introducidas.

3.4.3.- Configuración del escenario

A lo largo de este proyecto se utilizan distintas topologías de red, dependiendo del ataque objeto de estudio. Sin embargo, es posible realizar algunas configuraciones comunes a todas ellas:

- El controlador está en una red independiente. Con ello se asegura que con el controlador únicamente puede comunicarse el switch y no cualquier usuario de la red. Se define a esta red con la dirección 10.15.10.0/29.
- La dirección de partida de la red de los equipos es la 10.10.10.0/24, si bien esta cambia en función de los distintos escenarios a estudiar.
- El bridge interno del switch que se utiliza es el etiquetado como «br0».

3.4.4.- Configuración de elementos red

Con las indicaciones mostradas en el anterior apartado, se procede a la configuración de todos los elementos de la red.

3.4.4.1.- Controlador

Es el elemento central de la red y su correcta configuración es esencial. La instalación [34] se realiza de una manera muy sencilla por medio del gestor de paquetes de Python, **pip**.

```
$ pip install ryu
```

Para comprobar la instalación, se ejecuta a modo de prueba mediante el siguiente comando:

```
ryu run /usr/local/lib/python2.7/dist-packages/ryu/app/gui_topology/gui_topology.py
    simple_switch.py
```

Ello conllevará a la aparición de una larga lista de mensajes que finalizan con la siguiente sentencia: `wsgi starting up on http://0.0.0.0:8080`. Eso significa que está funcionando correctamente.

Una vez comprobada la correcta ejecución, se configura la interfaz de red con los parámetros descritos en el apartado 3.4.1. La dirección IP es la 15.0.0.1 y la máscara, al ser una red /29 la 255.255.255.248. Todo ello se realiza por medio de los comandos *ifconfig* de Linux:

```
$ ifconfig eth0 15.0.0.1 netmask 255.255.255.248 [up]
```

3.4.4.2.- Switch

En cada uno de los switches de la red se ha de especificar la dirección IP y el puerto del controlador o controladores, en el caso de cohabitar varios en el escenario de red. Esto se realiza por medio de un CLI que proporciona GNS3 vía Telnet. En Windows,

puede ser mediante un programa externo como PuTTY. Para MacOS se integra junto con la consola terminal del sistema operativo de Apple.

Se establece la configuración, en primer lugar, de la interfaz de red **eth0** que es, como se ha comentado en el apartado 3.4.1, la destinada para gestión. Se le asigna una IP en el rango 15.0.0.0/29. Al ocupar el controlador la dirección 15.0.0.1, va a ser la 15.0.0.2. Se utiliza de nuevo **ifconfig** como se muestra a continuación:

```
$ ifconfig eth0 15.0.0.2 netmask 255.255.255.248 [up]
```

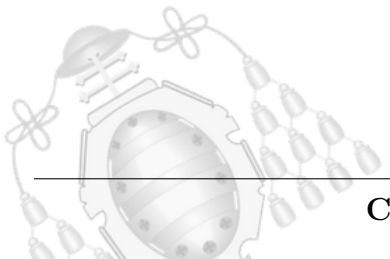
El siguiente paso es indicar la dirección del controlador y asignárselo a un puerto interno, para este caso el etiquetado como «br0». La dirección IP del controlador es la 15.0.0.1 y el puerto el 6635 TCP. Se utilizan comandos específicos de Open vSwitch para tal efecto

```
$ ovs-vsctl set controller br0 tcp:15.0.0.1:6635
```

3.4.4.3.- Resto de equipos

A lo largo de los diferentes escenarios que se van a realizar en este trabajo, pueden existir dos tipos de dispositivos que ejerzan como máquinas finales: bien como atacantes, como objetivo de los ataques, o simplemente emulación de equipos secundarios en la red, para comprobar la conectividad de la misma. Estos pueden ser:

- **Máquinas virtuales**: Son las que reciben los distintos ataques y todas ellas disponen de sistemas operativos Linux. Los parámetros de configuración de sus interfaces de red siguen el proceso análogo al descrito para el controlador.
- **VPCs**: Son máquinas con funcionalidades muy limitadas que simulan equipamiento de usuarios finales. Su propósito es comprobar la conectividad de red en distintos tipos de escenarios. Poseen una interfaz por línea de comandos. Para la asignación de direcciones IP basta con introducir el comando `ip` seguido de la dirección que se desee.



3.5.- Desarrollo de aplicaciones

Una vez configurado el escenario, es necesario mencionar la forma en que se van a desarrollar las aplicaciones para resolver problemas de seguridad en entornos de SDN. Como se ha ido comentando en el apartado 3.1.4 y sucesivos, existen en la actualidad multitud de ofertas de controladores de red, en las que cada uno lleva asociado un lenguaje de programación y una forma de desarrollar aplicaciones distinta.

Los entornos más extendidos son Java y C, liderados por ONOS y OpenDayLight. En los últimos años, han ido apareciendo diferentes alternativas en Python, como POX o Ryu entre otros grandes controladores. Existen también proyectos asociados a OpenDayLight, que hacen uso de este lenguaje de programación aunque no son del todo oficiales.

Para este proyecto, se explica la forma de desarrollar aplicaciones en el entorno de Ryu. Para ello, hace falta analizar la arquitectura base de toda aplicación.

3.5.1.- Arquitectura de una aplicación en Ryu

Existen varias aplicaciones que vienen de serie tras la primera instalación. Entre ellas, Firewalls, VLANs o SimpleSwitch. Se parte como base de SimpleSwitch, que es la aplicación base del resto.

Toda aplicación cuenta con una cola FIFO para recibir eventos. Estos pueden ser, por ejemplo, la llegada de un paquete al controlador o el cambio del estado en un puerto de un dispositivo. De igual manera que una aplicación recibe eventos, también puede generarlos: modificar tablas de flujo, cambiar el estado de un puerto, obtener estadísticas de tráfico, etcétera.

Cada evento recibido puede tener asociada una o varias funciones que realizan distintas tareas. El evento principal en todas ellas es la llegada de un paquete al controlador, evento **packet_in**. Para indicar este evento, se utiliza la marcación especial en Python con un @. A continuación se muestra un ejemplo de una función que maneje un paquete de llegada al controlador.

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packet_in_handler(self, ev):
   msg = ev.msg
   dp = msg.datapath
   ofp = dp.ofproto
   ofp_parser = dp.ofproto_parser
   actions = [ofp_parser.OFPActionOutput(ofp.OFPP_FLOOD)]
   out = ofp_parser.OFPPacketOut(
   datapath=dp, buffer_id=msg.buffer_id, in_port=msg.in_port, actions=actions)
   dp.send_msg(out)
```

En este ejemplo, cada vez que se produce un evento **OFPPacketIn** de llegada de paquete al controlador, se llama a la función **packet_in_handler**. A continuación, existen los siguientes elementos:

- **ev.msg**: Es un objeto que representa una estructura de datos de tipo **packet_in**.
- **msg.datapath**: Representa al switch origen del paquete.
- **dp.ofproto**: Es el protocolo OpenFlow negociado entre el controlador y el dispositivo (switch).
- **ofproto_parser**: Adecúa la estructura de cada paquete recibido en función de la versión del protocolo OpenFlow negociado. Es una función incluida en las librerías de Ryu.

El objeto **actions** representa el conjunto de acciones a tomar para ese determinado flujo. En este caso concreto, las acciones son OFPP_FLOOD, que quiere decir que se envía a partir de ahora por todos los puertos del dispositivo.

Por último, se prepara el mensaje a enviar al switch para que establezca el correspondiente flujo, en donde deben especificarse el switch, el puerto de ingreso y las acciones a tomar. Una vez construido, se envía de vuelta a dispositivo correspondiente.

También existen otros eventos presentes, por ejemplo la obtención de las características soportadas por el dispositivo, con un mensaje **features_request**. Dependiendo de la aplicación que se quiera desarrollar, son necesarios este u otros eventos.

El paquete recibido puede ser decodificado, por medio de las librerías packet o ethernet presentes. Para ello es necesaria su importación:

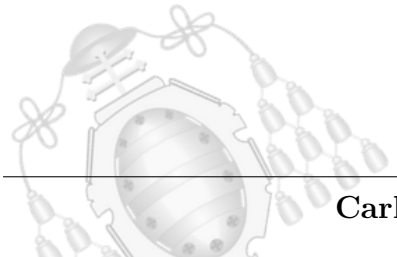
```
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
```

En el Anexo I se muestra al detalle el código de una aplicación simple, que consiste en un switch de capa 2.

3.5.2.- Ejecución de la aplicación

Las aplicaciones se guardan en un archivo de Python, cuya extensión es **py**. Una vez guardado, siguiendo el esquema de las aplicaciones preexistentes, se ejecuta con el comando Ryu para tal efecto:

```
ryu run /usr/local/lib/python2.7/dist-packages/ryu/app/gui_topology/gui_topology.py
    aplicacion.py
```



4. Aspectos de seguridad

Son múltiples los ataques que pueden producirse en una red LAN. En esta investigación se estudian algunos de ellos y la forma de poder detenerlos mediante aplicaciones en SDN. Si bien no se reproducen todos los distintos ataques existentes, se detallan a continuación los que son objeto de este trabajo y que no se pueden mitigar de manera sencilla en una red LAN tradicional sin invertir en hardware adicional, aplicar políticas de configuración muy estrictas en los dispositivos de la red o sin penalizar su rendimiento.

4.1.- Escaneo de servicios

La fase de reconocimiento es la primera de las múltiples que consta un ataque hacia una entidad o red. En ella se realiza una exploración inicial del ecosistema en el que se aloja un usuario con intenciones delictivas, con el objetivo de obtener la máxima información posible acerca de los dispositivos cercanos para, posteriormente, analizar sus potenciales vulnerabilidades con objeto de ser explotadas.

A su vez, esta fase de reconocimiento se puede subdividir en dos partes bien diferenciadas:

- **Fase de exploración inicial o *foot printing***: Todo tipo de trabajo de investigación, con el propósito de descubrir la mayor cantidad de información en relación a una empresa o entidad objetivo del ataque. Es, generalmente, información pública obtenida de forma totalmente legal por medio de vías de acceso general, por ejemplo una web corporativa.
- **Descubrimiento de servicios:** Se trata de obtener la mayor cantidad de información en la propia red a la que se tiene acceso y se quiere explotar. Estas técnicas pueden, a su vez, ser ejecutadas de dos maneras:

  - De forma pasiva: Si el usuario malicioso no tiene limitaciones de acceso físico, puede hacer acopio de diversas técnicas de obtención de información de

una forma no disruptiva. Herramientas de captación de tráfico en la red, como Wireshark o TCPDump, análisis de dispositivos o carpetas compartidas son algunas de las técnicas más comunes. Estos métodos no son eficaces si no se cuenta con una predisposición y un minucioso análisis de todo aquello de lo que se tenga acceso legítimo.

- De forma activa: Comprenden todo tipo de técnicas de escaneo y consultas en el sistema objetivo. A nivel de red, se corresponde con direcciones IP activas, nombres del dominio utilizados en el caso de redes con Directorio activo e incluso el nombre de los usuarios de la red. A nivel de sistema, todo lo que comprometa a puertos abiertos por los usuarios, aplicaciones en ejecución dentro de sus sistemas, o versiones de las mismas, son algunos ejemplos de información que puede obtenerse por medio de estas técnicas. Herramientas de escaneo en red como **nmap**, para sistemas Linux, son de las más completas y utilizadas en estos casos.

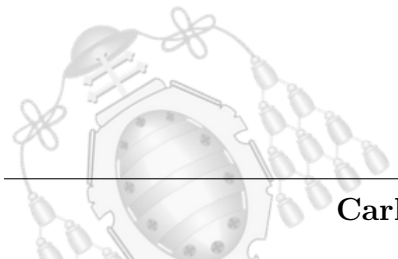
En lo que respecta a este proyecto, se hace hincapié en el descubrimiento de servicios de manera activa, tratando de localizarlos y aplicar las medidas más efectivas con el fin de detener posibles ataques en el entorno. Es una buena barrera de defensa la correcta aplicación de estas medidas, puesto que se tratan de los primeros pasos en un ataque.

4.1.1.- Escenario del ataque y procedimiento

Se parte de un escenario muy simple, en el que varios equipos están conectados a la misma red, y uno de ellos quiere realizar un escaneo, con el objetivo de identificar la mayor cantidad de información posible.

Existen multitud de herramientas para realizar este tipo de actos en redes. Para este proyecto, se hace uso del software nmap. Para llevar a cabo el escaneo, el usuario ejecuta un comando de escaneo en una red completa, como puede ser el que sigue:

```
$ nmap 192.168.0.0/24
```



4.1.2.- Mitigación del ataque en una red tradicional

En las redes convencionales la detección de escaneos en la red no es algo trivial. Son diversas las técnicas existentes, aunque todas ellas no están exentas de falsos positivos. Algunas técnicas, más proactivas, pasan por el apagado de puertos de los switches que no estén en uso para evitar conexiones de usuarios no autorizados. Otras más avanzadas consisten en la inclusión de sistemas de detección de intrusiones (IDS), programas que capturan y analizan todo el tráfico por medio de sondas virtuales ubicadas en distintos puntos de la red. Los IDS pueden o bien alertar al administrador de la presencia de alguna amenaza o también alertar y actuar sobre ella. Algunos ejemplos de IDS son Snort, Open DLP o Zeek.

También pueden aplicarse técnicas *anti-storming* en los switches de la red, filtrando el tráfico entrante en una interfaz para evitar *flooding* o inundación de paquetes. Sin embargo, estas técnicas no resultan del todo efectivas si los paquetes son muy espaciados en el tiempo.

4.1.3.- Mitigación del ataque en un entorno SDN

Para poder mitigar un escaneo en la red, se actúa directamente sobre el switch que da soporte a la red objeto de ese escaneo. Se va a partir de un enfoque reactivo, es decir, que una vez se detecte que un determinado usuario está realizando un escaneo, se realicen las acciones necesarias para detenerlo. Existen dos enfoques a la hora de detener un escaneo en red ejecutado por un usuario malintencionado:

- El primero, que sería el más brusco, es apagar el puerto en el switch al que está conectado el usuario que está realizando ese escaneo. De esta manera ya no tendría acceso a la red y el ataque quedaría paralizado.
- El segundo, y más elegante, es descartar todo el tráfico que provenga de ese usuario o puerto concreto. Con ello, se evita el apagado o encendido de puertos de manera constante y el atacante no es consciente de que ha sido detectado, puesto que la interfaz de red que le da conectividad sigue encendida. Para llevarlo a cabo, se establecen una serie de reglas en la tabla de flujos del switch.

Para este proyecto, se implementa principalmente el primer enfoque, dejando el segundo a futuras ampliaciones o mejoras de la aplicación. Como punto de partida, se debe caracterizar el tráfico que genera el programa de escaneo con el fin de poder ser detectado y actuar en consecuencia.

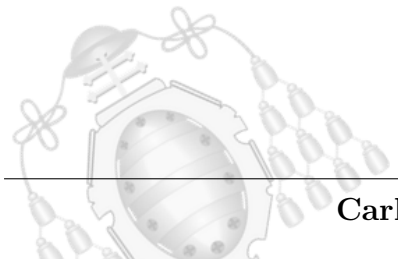
4.1.3.1.- Creación de la aplicación

A continuación se explican los pasos seguidos para la construcción de la aplicación en el entorno de las SDN para mitigar este tipo de ataques.

Ryu ofrece multitud de herramientas, módulos y ejemplos prácticos para llevar a cabo estas acciones. De aquí en adelante, se parte como base de una aplicación para manejar un switch mediante OpenFlow 1.3.

4.1.3.1.1.- Parametrización del tráfico malicioso

Por norma general, los programas de escaneo de servicios o equipos, en su fase de exploración, utilizan peticiones ARP sobre los posibles destinatarios. Una petición ARP consiste en obtener, a partir de una dirección lógica dada, su correspondiente dirección física. En ella se «pregunta» quién tiene una determinada dirección IP, siendo respondida por ese equipo con su dirección física. En la figura 4.1 se muestra el contenido de un paquete ARP capturado por el programa Wireshark.



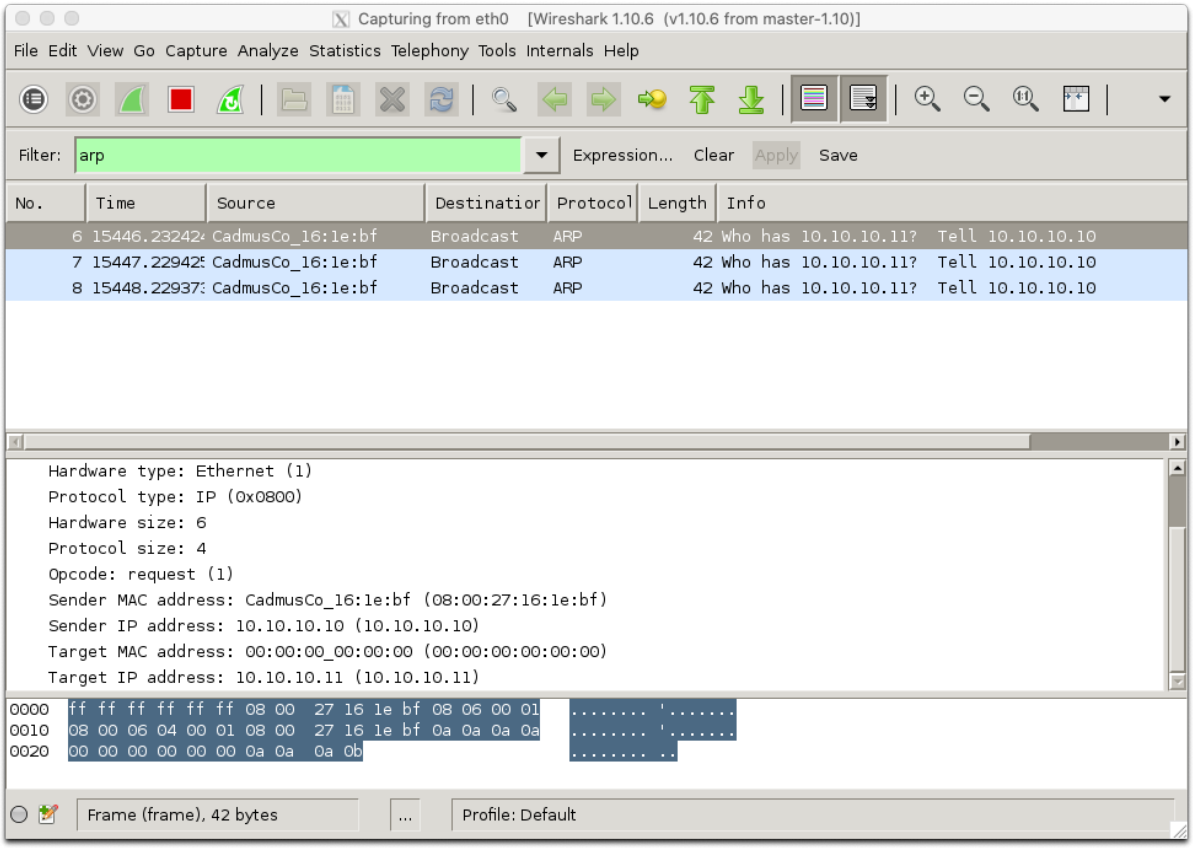


Figura 4.1.- Paquete ARP capturado por medio del programa Wireshark, donde la máquina con dirección IP 10.10.10.10 solicita quién tiene la IP 10.10.10.11.

Las direcciones IP pueden variar a lo largo del tiempo, dependiendo de si éstas son estáticas o dinámicas e, incluso, de la configuración establecida por el administrador de la red. Sin embargo, la dirección física es única para cada dispositivo del mundo y, por tanto, no varía. A partir de ellas puede determinarse información, por ejemplo, del tipo de dispositivo que se trata o de su fabricante.

En un entorno de red, en condiciones normales, estas peticiones también se utilizan cuando un administrador de un dispositivo cambia su dirección IP de forma manual. Éste envía, de manera automática, peticiones ARP con el objetivo de anunciarse a los demás dispositivos de la red y así ahorrar solicitudes por parte de los mismos.

4.1.3.1.2.- Implementación

Cada vez que el controlador recibe un paquete que no encaja en ninguna de las reglas que tiene implementadas, existen dos opciones a definir por el administrador de la red: o bien ese paquete es descartado, o se envía al controlador para su procesamiento. Esto es posible configurarlo gracias a lo que en OpenFlow se denomina *miss-table*, una de las novedades introducidas a partir de la versión 1.3. En el caso de las peticiones ARP, cada vez que un dispositivo la recibe, ésta es automáticamente enviada al controlador.

Por medio de distintas funciones que provee Ryu, es posible procesar este paquete y extraer de él todo tipo de información relevante para el propósito de este trabajo, entre otros:

- Dirección IP de origen y destino.
- Dirección MAC de origen y destino.
- Número del puerto del switch y dirección física por el que se ha recibido dicho paquete.

Para implementar la aplicación para detener un escaneo en red, es necesario establecer de antemano los criterios a seguir para identificar dichos ataques. Se toma como base la siguiente norma: si el número de peticiones ARP provenientes de una misma MAC de origen o puerto origen supera un umbral determinado, se procede a apagar dicho puerto, evitando así el escaneo.

Teniendo en cuenta un escaneo en una subred «tipo», esto es, con un máximo de 254 dispositivos, se presuponen un mínimo de otras tantas peticiones ARP. Es por ello que se determina un umbral de 50 peticiones ARP como máximo por dispositivo o puerto. En caso de que este umbral se supere, se procede al apagado del puerto correspondiente del dispositivo.

Al trabajar con los paquetes entrantes al controlador, toda la implementación se lleva a cabo dentro de la función **packet_in** definida en las aplicaciones por defecto de Ryu, tal y como se ha explicado en el apartado 3.5.1.

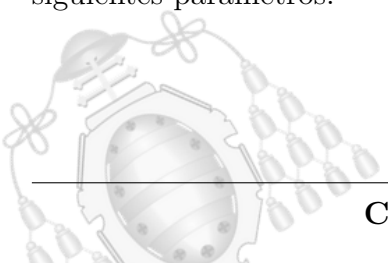
Para detectar si un paquete es ARP, en primer lugar se comprueba si la trama Ethernet recibida, en su campo EtherType se corresponde con el valor **0x0806**. Después, para ver si se trata de una petición y no una respuesta, basta con identificar si la dirección MAC de destino es «**ff:ff:ff:ff:ff:ff**». De cumplir estos dos criterios, se procede a identificar el puerto de origen de dicha petición, así como la dirección MAC del dispositivo que la ha originado. Una vez extraída dicha información, se debe registrar el número de veces que dicha petición ha llegado a través de ese puerto. Para ello, aprovechando el lenguaje de Python, se utiliza un diccionario. En él se puede almacenar, por medio de pares «clave» y «valor», todo el registro de puertos y el número de veces que cada uno ha registrado una petición ARP. Las claves han de ser identificadores únicos, que no deben repetirse bajo ningún concepto, por tanto se utilizan los números de los puertos del switch.

```
#Ejemplo de diccionario utilizado, donde se almacenan los registros de llegada de
    peticiones ARP por cada puerto
ports_dict =
          {
             "1": { "count": 2},#El puerto 1 ha recibido 2 peticiones ARP
             "3": { "count": 5} #El puerto 3 ha recibido 5 peticiones ARP
          }
```

Por cada petición ARP, se determina el puerto por el que ha llegado al switch y se comprueba si su número se encuentra en el diccionario como «clave». En caso de no ser así, se crea una nueva entrada en el diccionario y se le asigna un contador, con valor inicial de 1. En caso de existir dicha clave, este contador incrementa su valor de uno en uno cada vez que un paquete ARP sea recibido por dicho puerto.

Una vez se realiza esta comprobación, la siguiente a realizar es si dicho contador asociado a ese puerto ha superado el umbral establecido, que es de 50. De ser así, se ejecuta una instrucción para que el dispositivo apague ese puerto.

Para el apagado del puerto, se crea una función, **shutdown_port**, que recibe los siguientes parámetros:



- **datapath:** Objeto que representa el switch al que se le envía el mensaje de modificación del puerto.
- **port_no:** Número del puerto a apagar.
- **hard_addr:** Dirección física del puerto que se quiere apagar.

Internamente, en la función, se construye la petición **req**, en la que se adjuntan tanto el número de puerto, como la dirección física así como las instrucciones específicas de actuación sobre el puerto, esto es, el apagado del mismo, mediante la sentencia **OFPPC_PORT_DOWN**. Adicionalmente, se incluye el mensaje de advertencia **OFPP_PAUSE**, aunque no es obligatoria su inclusión. El resto de parámetros, son análogos a los indicados en el apartado 3.5.1. La función resultante se detalla en el código 4.1

Código 4.1.- Función creada para el apagado de un determinado puerto

```
def shutdown_port(self, datapath, port_no, hard_addr):
    ofp = datapath.ofproto
    ofp_parser = datapath.ofproto_parser
    req = ofp_parser.OFPPortMod(datapath, port_no, hard_addr, ofp.OFPPC_PORT_DOWN,
        datapath.ofproto.OFPPF_PAUSE)
    datapath.send_msg(req)
```

En el código 4.2 se detalla la parte de la aplicación que implementa la lógica de detección de escaneos. La aplicación completa se adjunta en el Anexo II.

Código 4.2.- Parte de código de la aplicación para detección de escaneos ARP

```
if eth.ethertype == ether_types.ARP:
    if dst == 'ff:ff:ff:ff:ff:ff':
        if in_port in ports_dict:
            ports_dict[in_port]['counter'] += 1
            if ports_dict[in_port]['counter'] >= 10:
                self.logger.debug("ALERTA: Escaneo de servicios detectado")
                self.shutdown_port(datapath, in_port, hard_addr)
                self.logger.debug("INFO: El puerto %s ha sido apagado",in_port)
            if dpid not in ports_dict:
                ports_dict[in_port]= {'counter': 1}
```

4.1.4.- Prueba de realización del ataque con éxito

La prueba es tan simple como ejecutar el software de escaneo de red **nmap**. Se procede a escanear toda la red a la cual se encuentra conectada la máquina atacante. Para obtener la información de la red, basta con teclear el comando «ifconfig» de tratarse un sistema operativo Linux o «ipconfig» en el caso de máquinas Windows. Ellos muestran la dirección IP de la máquina y la máscara de subred, detalles suficientes para conocer la red a la que se encuentra conectado.

En el escenario objeto de este estudio, la dirección base de red es la 10.10.10.0 y la máscara de subred es 255.255.255.0, por tanto alberga capacidad para 254 dispositivos. En este escenario se encuentra presente una única máquina, que es el objetivo del ataque y cuenta con la dirección IP 10.10.10.160. La ejecución del escaneo por nmap se realiza mediante la siguiente instrucción.

```
$ nmap 10.10.10.0/24
```

Ésta lanza el programa y, en unos pocos segundos, arroja toda la información descubierta del equipamiento presente en la red. Toda ella se muestra en la figura 4.2.

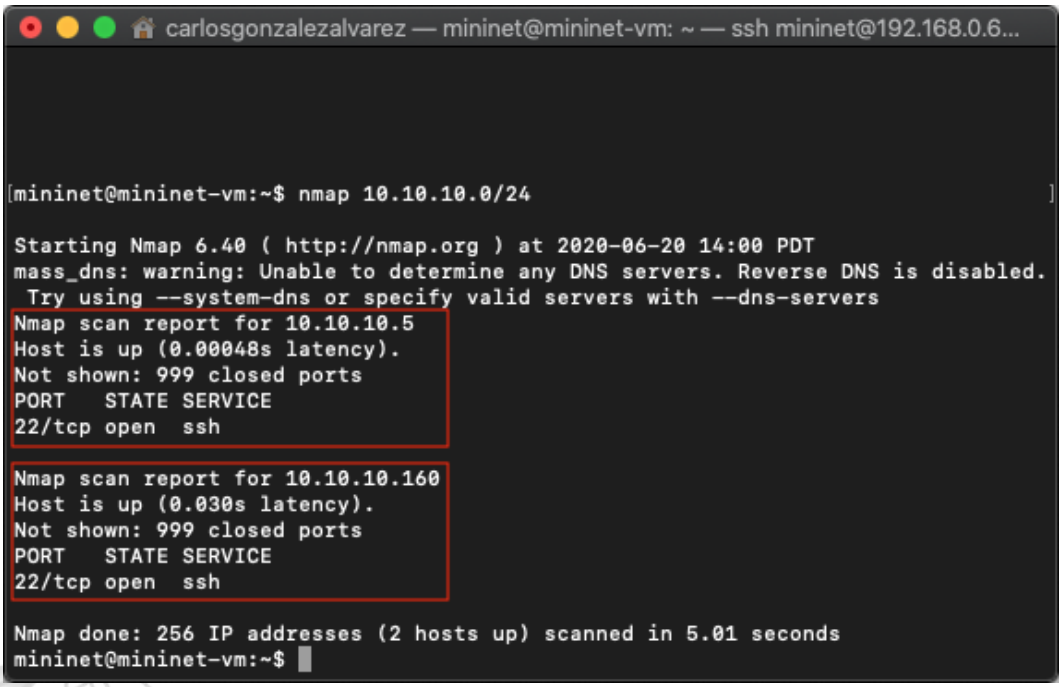


Figura 4.2.- Escaneo exitoso de la red usando el software nmap.

En la figura 4.2 se señalan dos servicios descubiertos. El superior se corresponde con la propia máquina que ejecuta el escaneo, cuya dirección IP asociada es la 10.10.10.5. El recuadro inferior muestra la información relativa a la segunda máquina, con dirección IP 10.10.10.160. Adicionalmente, se expone información relativa a los puertos que esa máquina tiene abiertos y sus protocolos asociados. Para ese caso particular, el puerto 22 TCP, correspondiente con el servicio de SSH.

En el controlador, la llegada de paquetes se muestra por consola con el mensaje de «EventOFPPacketIn», tal y como se constata en la figura 4.3.

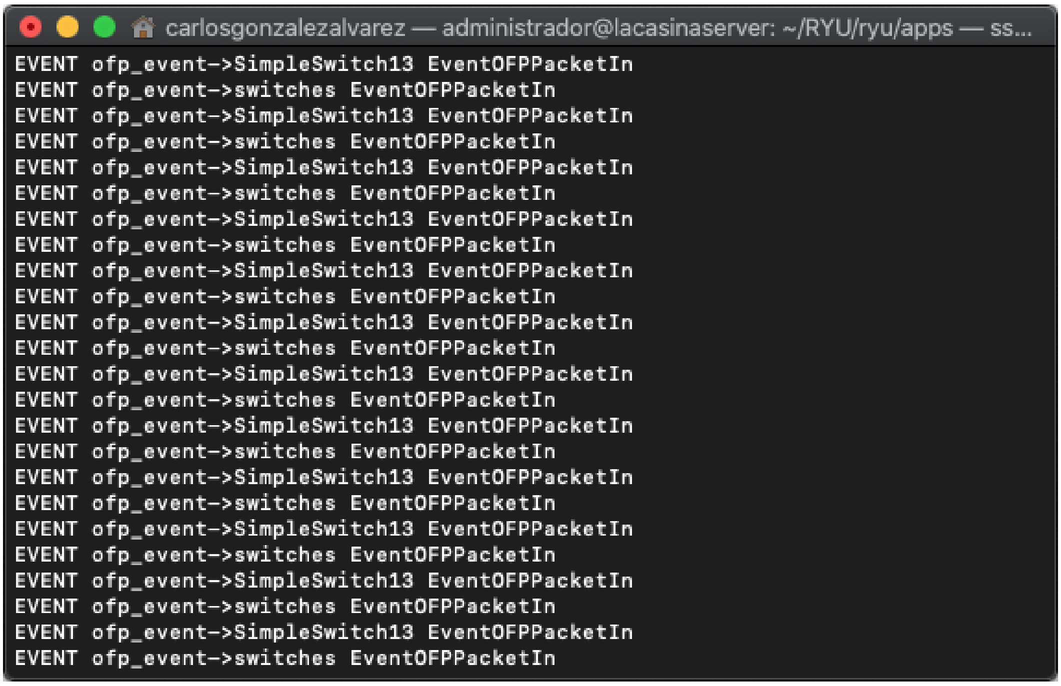


Figura 4.3.- Escaneo detectado por el controlador, en forma de llegada de paquetes ARP.

4.1.5.- Prueba de detención del ataque

Para hacer la comprobación de que los escaneos de red son detectados y se actúa en consecuencia, se habilita el código comentado en anteriores apartados, encargado de desactivar todo puerto que recibe un volumen de tráfico ARP superior a un umbral preestablecido. Cuando un puerto físico de un dispositivo cambia de estado, como es este caso de «encendido» a «apagado», se le envía un mensaje al controlador informando de que se han producido cambios en los puertos.

Se procede a realizar la misma operación que en el apartado anterior, ejecutando un escaneo en todo el conjunto de red para descubrir servicios activos. Si todo sale como está previsto, el controlador detecta el ataque y, en consecuencia, apaga el correspondiente puerto que daba servicio al atacante.

Este hecho es notificado por parte del controlador con mensajes de advertencia e información, que han sido puestos a propósito a la hora de desarrollar el programa. Dichos mensajes se observan en la figura 4.4. Se advierte al usuario de que se ha detectado un escaneo en el puerto 4 en este caso concreto. A continuación, se le informa de que dicho puerto se ha apagado. De manera adicional, se muestra un mensaje propio al protocolo OpenFlow, **DPSET**, informando de que se ha producido un cambio en uno de los puertos del switch, mensaje que se corresponde con la última línea mostrada en la citada figura 4.4.

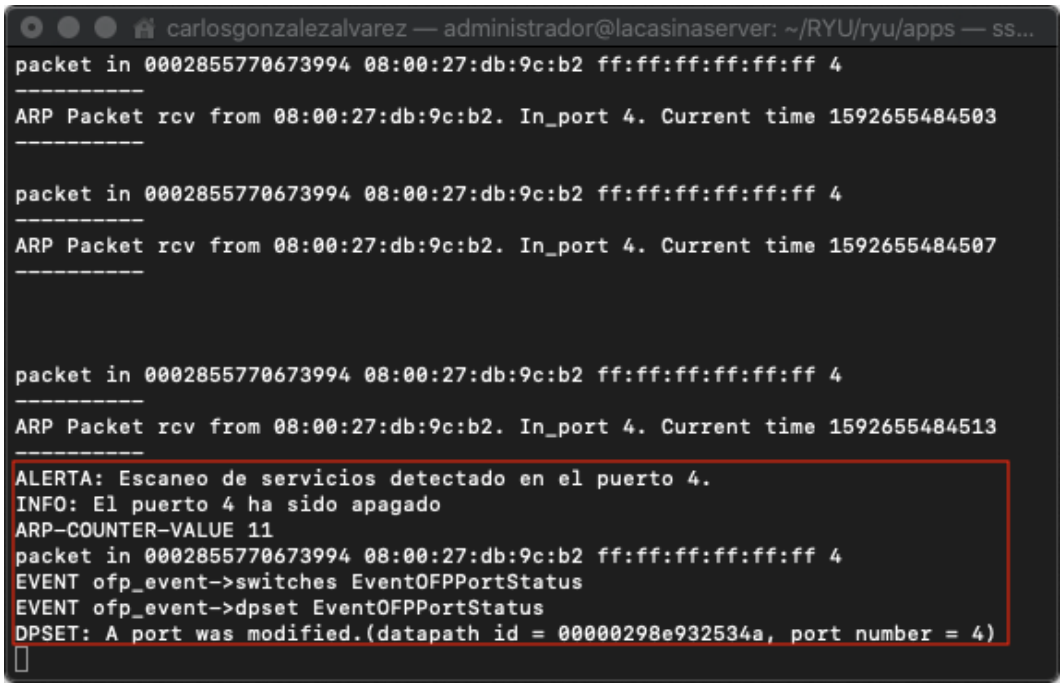


Figura 4.4.- Escaneo detectado por el controlador. Mensajes en forma de alerta del ataque y del apagado del puerto.

En consecuencia con lo anteriormente expuesto, por parte del atacante no debiera ser posible llevar a cabo con éxito el escaneo de la red. En los resultados arrojados por el programa nmap aparece como servicio activo, únicamente, el propio terminal desde

el cual se ejecuta el ataque. Esto se refleja en la figura 4.5. Sin embargo, debe hacerse aquí una mención: si la dirección IP hubiese sido, por ejemplo, 10.10.10.25, el escaneo sí daría resultados positivos, puesto que el límite establecido de peticiones es de 50, aunque el resultado final sería el mismo: el puerto del atacante apagado y, por tanto, el ataque detenido.

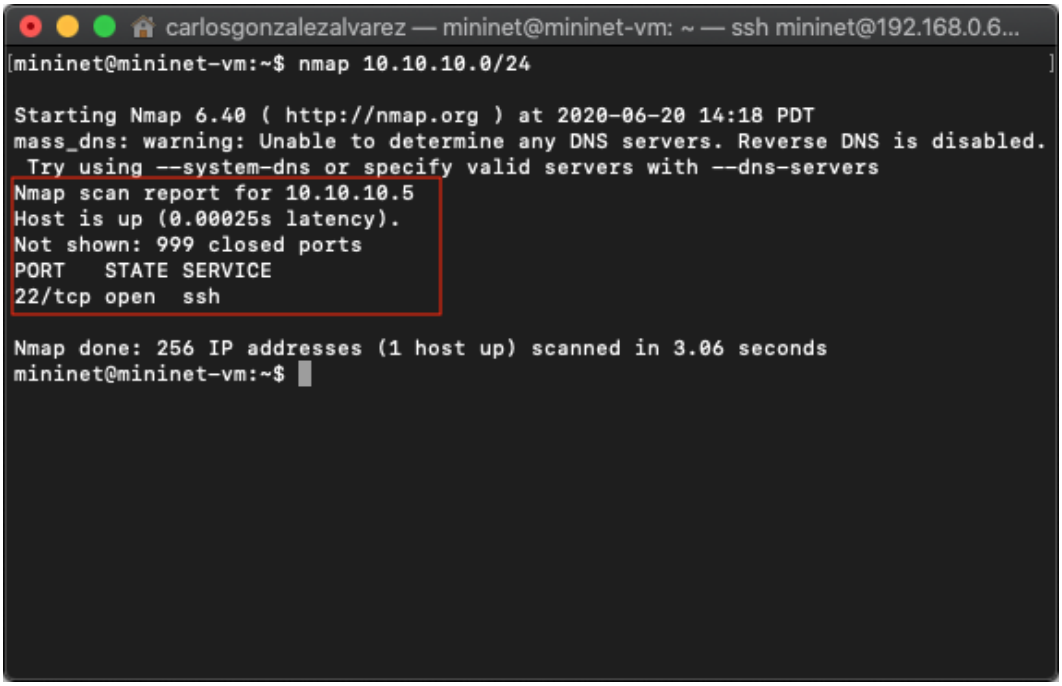


Figura 4.5.- Escaneo no exitoso de la red usando el software nmap.

Como comprobación final, en la máquina atacante, se comprueba la conectividad con la red mediante la realización de un simple ping a la dirección de la máquina objeto de ataque, la 10.10.10.160. Se observa claramente cómo no es posible alcanzar dicho objetivo, signo de que la comunicación con la red ha sido bloqueada, como se muestra en la figura 4.6.

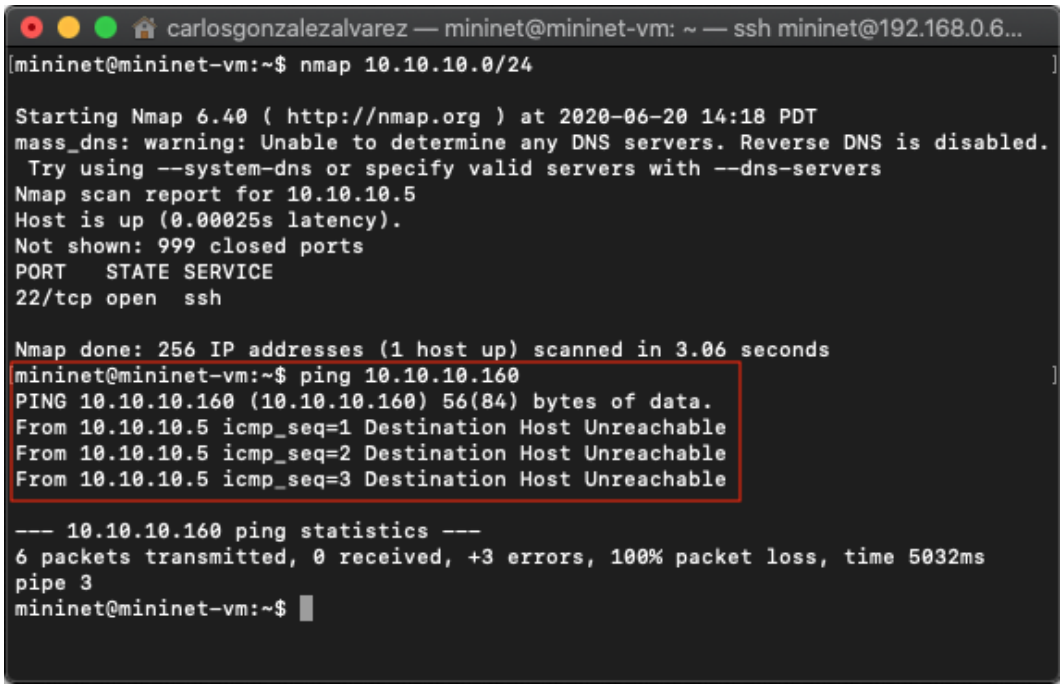


Figura 4.6.- Ping realizado tras detectar ataque, no funciona por el apagado del puerto.

Se determina, por tanto, el éxito de las pruebas realizadas. El controlador detecta un escaneo de servicios ilegítimo y, en consecuencia, actúa según los criterios establecidos bloqueando dicho puerto de comunicaciones. Ha de tenerse en cuenta también, como se ha expuesto anteriormente, el hecho de que el límite de peticiones ha sido establecido en 50.

4.1.6.- Mejoras

Toda aplicación es susceptible de mejoras. El criterio utilizado para el apagado de un puerto es el número de veces que ha recibido una petición ARP. Sin embargo, este contador en bruto puede considerar como falsos positivos simples cambios en la dirección IP realizada por una determinada máquina, o peticiones rutinarias realizadas por algún determinado programa instalado y legítimo.

4.1.6.1.- Detección del ataque por volumen de tráfico

Una de las características de este tipo de escaneos en red es el volumen de peticiones ARP por unidad de tiempo. En las figuras 4.2 y 4.5 puede observarse como se ha empleado una media de 3 a 5 segundos para sendos escaneos en una red de 256 dispositivos. Esto supone enviar una media de 85 peticiones por segundo. Un volumen de tráfico bastante significativo, que implica que el controlador reciba un paquete cada 12 milisegundos.

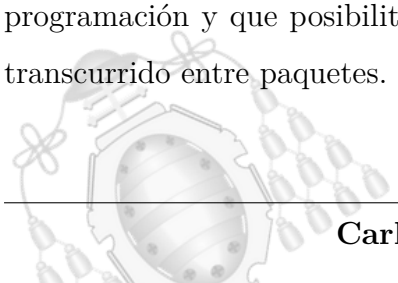
Teniendo esto en cuenta, en caso de que se reciban 50 peticiones ARP por un mismo puerto, se pueden distinguir dos situaciones bien distintas:

- Por un lado, que esas 50 peticiones hayan sido espaciadas a lo largo del tiempo. Se trataría de una situación normal dentro de cualquier red, fruto de ajustes, instalación de programas o apagado y encendido rutinario de equipamiento en red.
- Que se hubieran recibido en un espacio muy corto de tiempo, entre medio minuto y un minuto. En este caso, se hablaría más bien de un escaneo de servicios.

Es por ello que se evitarían falsos positivos por ataques, debido al recuento total de paquetes sin tener en cuenta la diferencia de llegada entre ellos.

Esto, trasladado a la aplicación desarrollada, implica que se debe tener en cuenta no solamente el tipo de paquete recibido en cada puerto, sino la diferencia de tiempo existente entre el actual paquete y el último recibido.

El proceso de registro en cada llegada de un paquete ARP sigue siendo, en esencia, el mismo que se ha explicado en el apartado 4.1.3.1.2, pero introduciendo pequeños cambios. En el diccionario de registro de paquetes, para cada puerto, se añade un segundo campo, denominado **last_timestamp**. En él, se introduce el tiempo en formato Unix, en milisegundos, una medida muy extendida en el mundo de la programación y que posibilita obtener a partir de ella, y de forma precisa, el tiempo transcurrido entre paquetes.



```
#Ejemplo de diccionario utilizado, donde se almacenan los registros de llegada de
    peticiones ARP por cada puerto y el tiempo de llegada del ultimo paquete
    recibido en dicho puerto
ports_dict =
{
"1": {
      "last_timestamp": 15124645, #Tiempo de llegada del ultimo paquete
      "count": 5 #Si el tiempo entre paquetes es inferior a un umbral, se
          incrementa el contador
      },
"3": {
      "last_timestamp": 15124621,
      "count": 2
      },
}
```

Cuando llegue un nuevo paquete ARP se hacen dos comprobaciones: la primera, si existe algún registro de ese puerto en el diccionario. De ser así, se mira el último tiempo de llegada y se calcula la diferencia con la marca de tiempo actual. Si ésta es igual o inferior a un umbral establecido, como base 50 milisegundos, se incrementa en una unidad el contador. A continuación, se actualiza el valor del campo **last_timestamp** con el valor de tiempo actual.

De superarse para ese puerto un valor preestablecido en su contador se procede, de la misma forma que en la aplicación original, al apagado del citado puerto.

4.1.6.2.- Filtrado del tráfico

En esta variante de la aplicación, se aplica una política distinta para detener el escaneo en la red. En vez de proceder al apagado del correspondiente puerto, se adjunta una regla en la tabla de flujos del dispositivo que deniegue, para dicho puerto, todo tráfico entrante. Esto ataja de una forma silenciosa e imperceptible para el atacante el problema. El resultado son escaneos sin dispositivos asociados, manteniendo en todo

momento la conectividad del usuario atacante, sin darle ningún tipo de pista de que haya sido descubierto.

Para insertar la regla en la tabla de flujos, se crea una función dentro de la clase general de la aplicación. Esta contiene todas las instrucciones necesarias, siguiendo la documentación establecida por Ryu. En esta función se pasan como parámetros los siguientes objetos:

- **datapath**: Es el objeto que hace referencia al switch que el controlador está orquestando. De él se extraen dos variables: *parser*, que adecúa la estructura de los mensajes en base a la versión de OpenFlow que se utiliza, y *ofproto*, que define qué versión de OpenFlow está siendo utilizada.
- **Port**: Puerto de ingreso del paquete en el dispositivo de red.

Tal y como establece la documentación de Ryu, en lo referente a creación y aplicación de flujos sobre dispositivos, se crea un objeto **OFPFlowMod** que contenga, entre otros, los siguientes objetos:

- **match**: El conjunto de reglas de flujo que deben satisfacer los paquetes que procesa el switch. Aquí se especifica que todo aquel paquete procedente de ese puerto sea descartado.
- **Instructions**: Las acciones a realizar sobre esos paquetes que satisfacen el criterio anteriormente descrito. En este caso concreto, se quiere que todos ellos sean descartados por el switch. Esto se indica con la variable **OFPIT_CLEAR_ACTIONS**.
- **out_port**: Puerto o conjunto de puertos de salida sobre los que aplicar esta regla. No se hacen distinciones de puertos en esta primera versión de la regla, por tanto se indica que se aplique a todos los puertos del dispositivo. Esto se indica con la opción **OFPP_ANY**.
- **out_group**: Los grupos son abstracciones de OpenFlow que permiten representar un conjunto de puertos como una única entidad. Un ejemplo de grupos es el de multicast. Aquí tampoco se hacen distinciones y se marcan todos

los grupos con la opción **OFPG_ANY**. La función resultante se muestra en el código 4.3.

Código 4.3.- Función creada para descartar tráfico entrante de un determinado puerto

```
def drop_arp_traffic(self,datapath,port):
    parser = datapath.ofproto_parser
    ofproto = datapath.ofproto
    match = parser.OFPMatch(in_port=port)
    actions = [parser.OFPInstructionActions(ofproto.OFPIT_CLEAR_ACTIONS,[])]
    mod = parser.OFPFlowMod(datapath=datapath,
    out_port=ofproto.OFPP_ANY,
    out_group=ofproto.OFPG_ANY,
    match=match, instructions=actions)
    datapath.send_msg(mod)
```

4.1.6.3.- Discriminación selectiva de puertos

Hasta el momento, la política de análisis de tráfico no discriminaba entre puertos, y se aplicaba de manera uniforme a todos ellos. Existen configuraciones de red que, por distintos motivos, pueden provocar la llegada de una «tormenta» de peticiones ARP. Por ejemplo, que tras un determinado puerto se encuentre una red formada por múltiples dispositivos y que un encendido simultáneo de los mismos provoquen grandes volúmenes de tráfico. También podría darse la casuística de que un puerto del dispositivo esté asignado para un programa cuya función sea, explícitamente, realizar escaneos de red. Se debe, por tanto, excluir a dichos puertos de la aplicación de las políticas de apagado de puertos o filtrado de tráfico.

Para aplicar esta política selectiva, basta con la introducción de una variable que contenga el identificador único de los puertos a los cuales no se quiere aplicar la política de detección de escaneos en red. Una vez incluida dicha variable, en las comprobaciones de llegada de paquetes ARP, se realiza una primera comprobación de si el puerto de llegada de dicho paquete se encuentra excluido o no.

En términos de rendimiento, no es la solución más adecuada. Permitir en puertos específicos la llegada de tráfico ARP masivo implica que todo éste se redirija

al controlador y sea convenientemente procesado. Esto penaliza notablemente el rendimiento del controlador. Es por ello que, como medida adicional, es conveniente añadir una regla en la tabla de flujos del dispositivo que no redirija todo el tráfico ARP recibido por dicho puerto al controlador.

4.1.6.4.- Fichero de configuración

Hasta ahora, todos los parámetros de las aplicaciones mencionadas con anterioridad estaban definidos explícitamente en el código fuente. Esta técnica, denominada *hard-coding* en inglés, está altamente desaconsejada en términos de programación, máxime si se trata de variables que contengan información sensible como usuarios y contraseñas. Es necesario llevar, por tanto, todos los parámetros susceptibles de ser modificados por cualquier administrador a un fichero de configuración externo.

Son muchos los distintos tipos de formatos disponibles para ficheros de configuración. Uno de los más extendidos es el formato YAML, un lenguaje basado en algunos muy extendidos como XML, Python, Perl o HTML. Su sintaxis es muy sencilla y su diseño está pensado para ser un lenguaje muy legible.

Este fichero de configuración se ubica en la misma carpeta que contiene la aplicación, y cuenta con la siguiente estructura:

- **Deltha_time**: El tiempo máximo permitido entre peticiones ARP, en milisegundos.
- **Counter**: El contador máximo a partir del cual activar las políticas de mitigación de escaneos en red.
- **Allowed_ports**: Lista que contiene el número de los puertos sobre los que no aplicar las políticas.
- **Enable_traffic_filter**: Campo en el que se indica si activar el filtrado de tráfico en vez del apagado de puertos.

Todo ello, estructurado en un archivo YAML, quedaría como sigue en el código 4.4.

Código 4.4.- Fichero de configuración para aplicación de detección escaneos en red

```
- ARP_MODULE:
  - Deltha_time: 50 #In milliseconds
  - Counter: 50
  - Allowed_ports: [1, 3, 4]
  - Enable_traffic_filter : False
```

4.2.- Ataques PVLAN

Las PVLAN, o VLAN privadas, son subdivisiones de una VLAN. De un modo coloquial, se podrían entender como «las VLAN de las VLAN», aunque con ciertos matices.

Este concepto introduce algunos cambios respecto a las VLAN tradicionales en lo que a división del dominio de difusión se refiere. Existen tres tipos de PVLAN [46], y cada una tiene su propio conjunto de reglas:

- **Promiscua**: Un equipo que esté conectado a un puerto en modo «promiscuo», tiene conectividad con cualquier otro equipo de la VLAN.
- **Aislada**: Solamente permite la conectividad con aquellos dispositivos conectados a un puerto promiscuo, nunca con otros que estén conectados a uno aislado.
- **Comunitaria**: Permiten la comunicación tanto con otros puertos de la misma comunidad como aquellos que se encuentren en modo promiscuo.

4.2.1.- Escenario del ataque

Se parte de una configuración de red tal y como se representa en la figura 4.7, compuesto por:

- Un router, que actúa como la puerta de enlace de la red (10.10.10.120) con dirección mac **AA:AA:AA:AA:AA**.
- Un PC, que es el atacante, y pertenece a la PVLAN 1 (10.10.10.1).
- Otro PC, objetivo del atacante y perteneciente a la PVLAN 2 (10.10.10.2).
- El switch que da conectividad a los equipos.

- El controlador de la red.

Se introduce en el escenario un nuevo elemento, en este caso, la puerta de enlace. Para ello, se utiliza la imagen de un router de Cisco, modelo c7200. Se configura la puerta de enlace con los siguientes parámetros:

- Dirección IP: 10.10.10.120
- Máscara de subred: 255.255.255.0

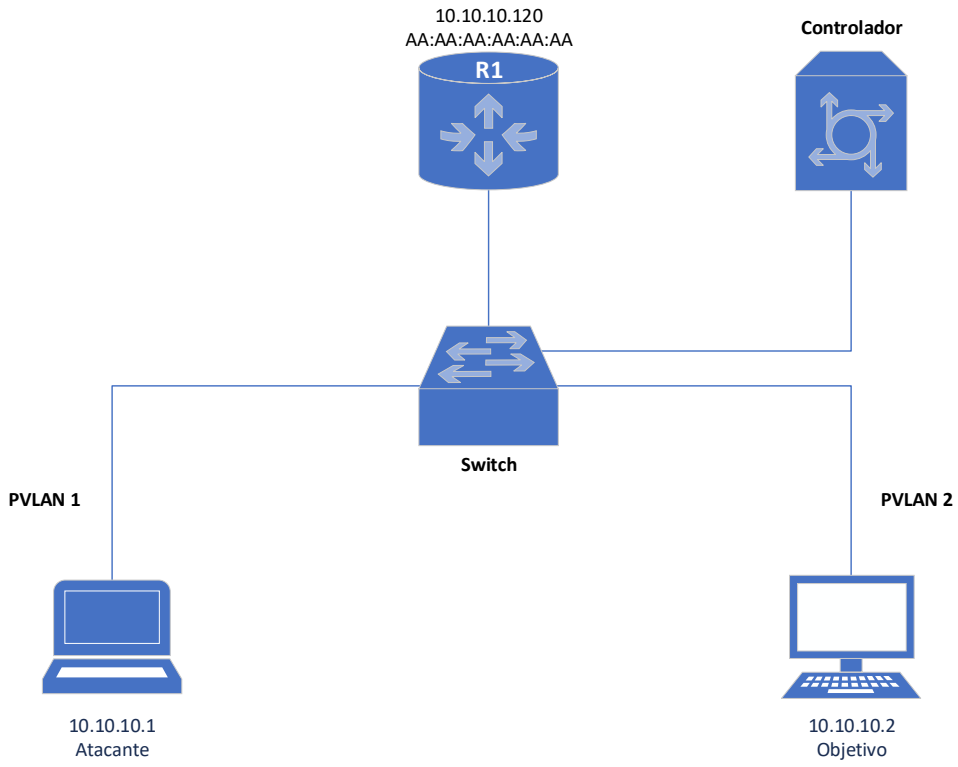


Figura 4.7.- Escenario para el ataque de PVLAN.

4.2.2.- Procedimiento del ataque

El objetivo es atacar a una máquina con la que, en principio, no se permite la comunicación. La conectividad entre redes PVLAN privadas no es posible, sin embargo ambas sí pueden comunicarse con la puerta de enlace. El atacante genera un paquete fraudulento con una dirección IP de destino la del objetivo del ataque, pero con dirección MAC de la puerta de enlace. El paquete es recibido por el switch y lo envía a la puerta de enlace correspondiente, puesto que su MAC está presente en dicho

paquete. Una vez llega al router, analiza la dirección IP de destino, con la cual sí tiene comunicación, y es enviada al usuario objetivo del ataque.

Se trata de una ataque unidireccional y puede utilizarse para realizar denegación de servicios o inyección de malware vía UDP, entre otros. Por tanto, para generar el paquete fraudulento el atacante debe generar una trama con los siguientes campos:

- **IP de origen**: La suya propia.
- **IP de destino**: La IP de la máquina a atacar.
- **MAC de origen**: La suya propia.
- **MAC de destino**: La de la puerta de enlace (router).

4.2.3.- Mitigación del ataque en una red tradicional

Para evitar este tipo de ataques en el plano de las redes convencionales, se actuaría sobre el router por medio de la declaración de listas de control de acceso (ACLs). En ella se especifica que todo el tráfico entre máquinas de la misma red debe ser bloqueado. En la figura 4.8 se muestra un ejemplo de una ACL para un router de Cisco, que deniega todo el tráfico IP entre dos máquinas de distintas PVLAN, la máquina 10.10.10.1 y la máquina 10.10.10.2.

Figura 4.8.- Lista de control de acceso para un router de Cisco, con el objetivo de denegar todas las comunicaciones IP entre dos máquinas.

4.2.4.- Detención del ataque en un entrono de SDN

En el terreno de las redes definidas por software, es posible actuar directamente sobre el switch, estableciendo el mismo criterio que en las listas de control de acceso

pero sobre una tabla de flujo: que todo tráfico entre equipos de una misma red se bloquee.

Como bien se ha explicado en la sección 2.4, para cada flujo de tráfico entrante en el switch se puede establecer una serie de acciones a tomar por el dispositivo, a destacar tres: el envío por un determinado puerto del switch, el envío de este tráfico al controlador o su descarte. En este caso, se establece una regla que indique que todo tráfico entre dispositivos de la misma red se descarte. Esta regla sirve como base, sujeta a futuros ajustes en función de nuevos requerimientos que puedan surgir.

OpenFlow, entre sus reglas para las tablas de flujo, permite establecer como criterios tanto la dirección IP de origen como la de destino. Para una red con múltiples dispositivos a los que queremos aplicar esta directriz, implica crear una regla por cada uno de ellos. Es necesario buscar alguna forma de aplicar el menor número de reglas posibles que afecten al mayor conjunto de dispositivos, por ejemplo, especificar una subred y no solamente un único dispositivo. Esto es importante, puesto que el número de entradas en la tabla de flujos es limitado.

4.2.4.1.- Creación de la aplicación

En el plano del controlador objeto de este proyecto, Ryu, se debe crear la aplicación que introduzca automáticamente la regla, o reglas, en la tabla de flujos del switch, con el fin mitigar este tipo de ataques.

La instrucción que ha de darse al switch debe indicar que toda petición cuya dirección de origen y de destino se encuentren en el mismo segmento de red, sea descartada. Aunque parezca una regla sencilla, se debe lidiar con determinados aspectos a la hora de implementarla correctamente, entre ellos:

- A priori, la dirección de la red y de la puerta de enlace son desconocidas. Esto es importante a la hora de establecer una regla en la tabla de flujos, que sirva para todos los equipos.
- OpenFlow permite establecer como criterios en las tablas de flujo, entre otros campos, las direcciones IP origen y destino de los paquetes. Es objeto de

investigación si dichas reglas permiten especificar un conjunto de equipos o una red entera con una única regla.

- De conocerse todo lo expuesto anteriormente y ser viable, se ha de encontrar su traducción correspondiente al lenguaje de Ryu para su correcta inclusión en la tabla de flujos del Switch.

4.2.4.1.1.- Obtención de parámetros de la red

Para crear las reglas de flujos en el switch, se deben de indicar las direcciones de red sobre las que se pretende restringir el tráfico. No se trata de una información trivial, puesto que cada red es diferente y esta información varía dependiendo de la configuración realizada. Es por ello que, dicha información, ha de ser suministrada por el propio administrador de la red.

De manera oficial, dichos parámetros se introducen en un fichero de configuración cuyo formato se determina mas adelante. Este fichero es procesado por la aplicación y, a partir de los datos obtenidos, se aplican las correspondientes reglas. En él se especifican los valores siguientes:

- Dirección base de la red.
- Máscara de subred.
- Dirección IP de la puerta de enlace.
- Dirección MAC de la puerta de enlace.

A falta de perfilar y determinar la implementación del fichero de configuración, así como su formato, estos parámetros inicialmente se incluyen en el propio código fuente de la aplicación, a modo de realizar las correspondientes pruebas.

4.2.4.1.2.- Implementación de la regla

Antes de proceder a ello, queda pendiente resolver una duda planteada en el apartado 4.2.4.1, y es ver si OpenFlow 1.3, dentro de su conjunto de reglas, permite la especificación de una subred en sus criterios de IP tanto de origen como de destino. La

información esgrimida por Ryu establece que esto es posible, siempre que se indique la dirección base de la red así como su máscara, en formato decimal.

En el código de la aplicación, se sitúan las instrucciones que añaden las reglas al switch dentro de una función de la clase principal, bajo el nombre de «add_flow_pvlan».

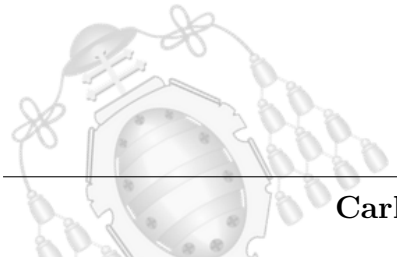
En la función, se pasan como parámetros los siguientes objetos:

- **datapath**: Es el objeto que hace referencia al switch que el controlador está orquestando. De él se extraen dos variables: *parser*, que adecua la estructura de los mensaje en base a la versión de OpenFlow que se utiliza y *ofproto*, que define qué versión de OpenFlow utilizada.
- **Address**: Dirección base de la red en la que se quiere evitar los ataques PVLAN.
- **subnet_mask**: Máscara de subred.

Tal y como establece la documentación de Ryu, en lo referente a creación y aplicación de flujos sobre dispositivos, se crea un objeto **OFPFlowMod**. El resultado final de la función es el que se muestra en el código 4.5

Código 4.5.- Función para creación de regla PVLAN

```
def add_flow_pvlan(self,datapath,address,subnet_mask):
    parser = datapath.ofproto_parser
    ofproto = datapath.ofproto
    match = parser.OFPMatch(eth_type=ether_types.ETH_TYPE_IP,
        ipv4_dst=(address, subnet_mask), ipv4_src=(address, subnet_mask))
    actions = [parser.OFPInstructionActions(ofproto.OFPIT_CLEAR_ACTIONS,[])]
    mod = parser.OFPFlowMod(datapath=datapath,
    out_port=ofproto.OFPP_ANY,
    out_group=ofproto.OFPG_ANY,
    match=match, instructions=actions)
    datapath.send_msg(mod)
```



4.2.5.- Prueba de realización del ataque con éxito

Antes de poner a prueba la aplicación es necesario, en primer lugar, comprobar que en el entorno de red es posible realizar este ataque. Para ello, se disponen de las siguientes utilidades:

- Una herramienta de generación de paquetes en la máquina atacante, que permita generar el tráfico malicioso con el fin de llegar al objetivo. Esto es posible gracias al software **Scapy**.
- En la máquina objetivo del ataque se hace uso de una herramienta para analizar todo el tráfico entrante y poder comprobar que los paquetes han llegado y, por tanto, que el ataque ha sido realizado con éxito. Se utiliza para ello Wireshark.

La ejecución de Scapy se realiza vía comandos, haciendo uso de su propia consola basada en Python. Sin embargo, la máquina destino no cuenta con interfaz gráfica para el uso de Wireshark. Existen varias soluciones a este inconveniente:

1. Utilizar en su defecto TCPDump, herramienta incluida en todas las distribuciones de Linux y la cual no precisa de interfaz gráfica.
2. Usar Wireshark en su versión por línea de comandos (*tshark*)
3. Hacer uso del protocolo X11 por medio de SSH que permite ejecutar, de forma gráfica, aplicaciones de Linux.

Se utiliza esta última opción debido a que Wireshark, en su versión gráfica, resulta un programa muy sencillo de utilizar.

Para la generación de los paquetes en Scapy, se ha de indicar la dirección MAC de destino, en este caso la de la puerta de enlace. La IP de destino es la de la máquina objetivo del ataque y, la de origen, la del atacante. También se especifica la interfaz por la cual enviar el paquete, en este caso, la etiquetada como «eth1». El paquete se configura de acuerdo a las especificaciones mostradas en la tabla 4.1 y es generado mediante el código 4.6.

Campo	Valor
MAC Origen	00:00:00:00:02:00
MAC Destino	ca:01:13:13:00:00
IP Origen	10.10.10.5
IP Destino	10.10.10.6
Interfaz	eth1

Tabla 4.1.- Datos de configuración para generación del paquete de ataque a una PVLAN en Scapy.

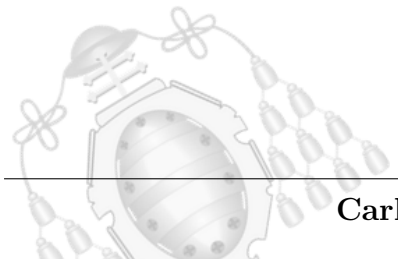
Código 4.6.- Generación de paquete para ataque PVLAN

```
>>> sendp(Ether(src="00:00:00:00:02:00",
    dst="ca:01:13:13:00:00")/IP(dst="10.10.10.6",ttl=(1,4)), iface="eth1")
```

Cuando se haya generado correctamente el paquete, se inicia el capturador de tráfico en la máquina objetivo del ataque. Se hace uso de XQuartz, la herramienta de X11 para sistemas operativos de MacOS. El login se realiza por medio de SSH a la máquina y, posteriormente, se ejecuta Wireshark con permisos de administración, por medio del siguiente comando:

```
$ sudo wireshark
```

Una vez iniciado, se selecciona la interfaz de red a través de la cual se van a capturar los paquetes. En el caso de esta máquina, la etiquetada como «eth0», tal y como se muestra en la figura 4.9



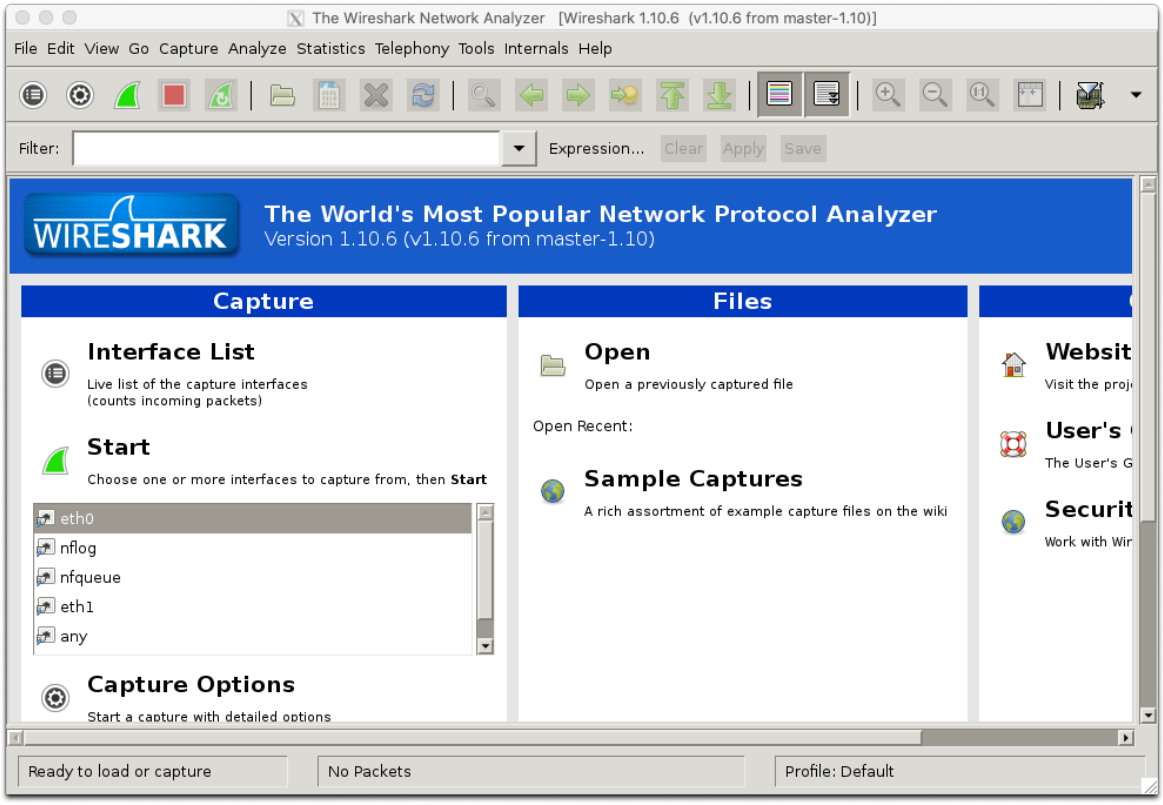


Figura 4.9.- Configuración de Wireshark para capturar paquetes de la interfaz Eth0.

El siguiente paso es ejecutar Scapy en la máquina atacante. Si todo transcurre como se espera, debería aparecer un mensaje en la consola indicando que los paquetes (cuatro en este caso) se han enviado exitosamente, hecho que así sucede tal y como se puede observar en la figura 4.10.

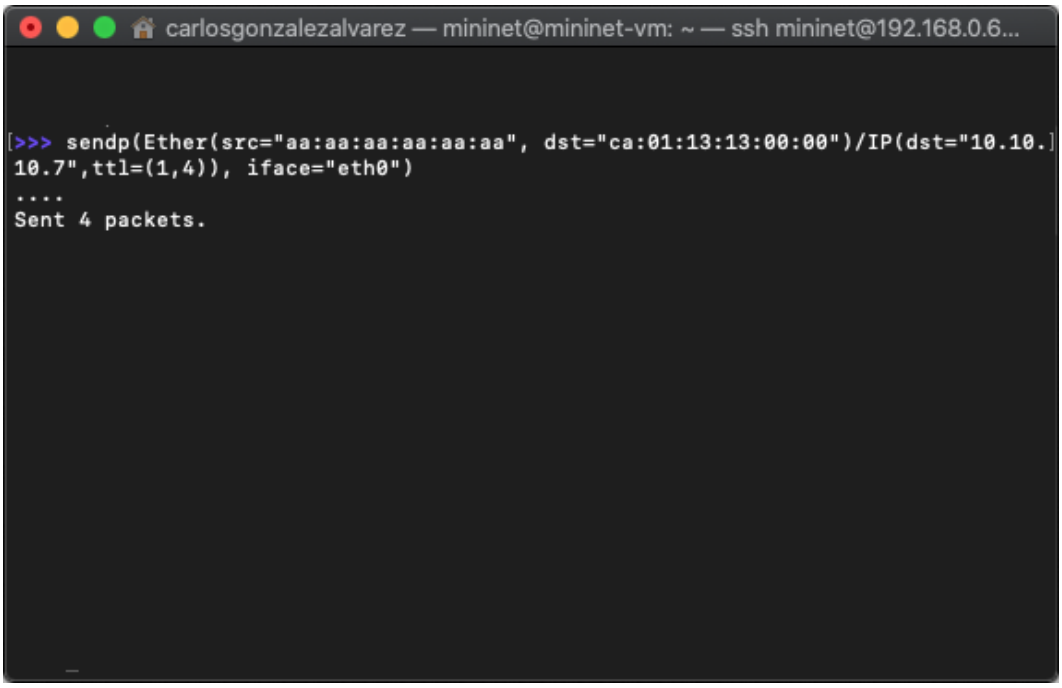


Figura 4.10.- Envío de paquetes exitoso desde la interfaz por línea de comandos de Scapy.

De igual manera, si todo sale como estaba previsto, se debe encontrar en el objetivo del ataque el rastro del ataque, en forma de paquetes recibidos. En la figura 4.11 se observa perfectamente este hecho, con los paquetes marcados en rojo.

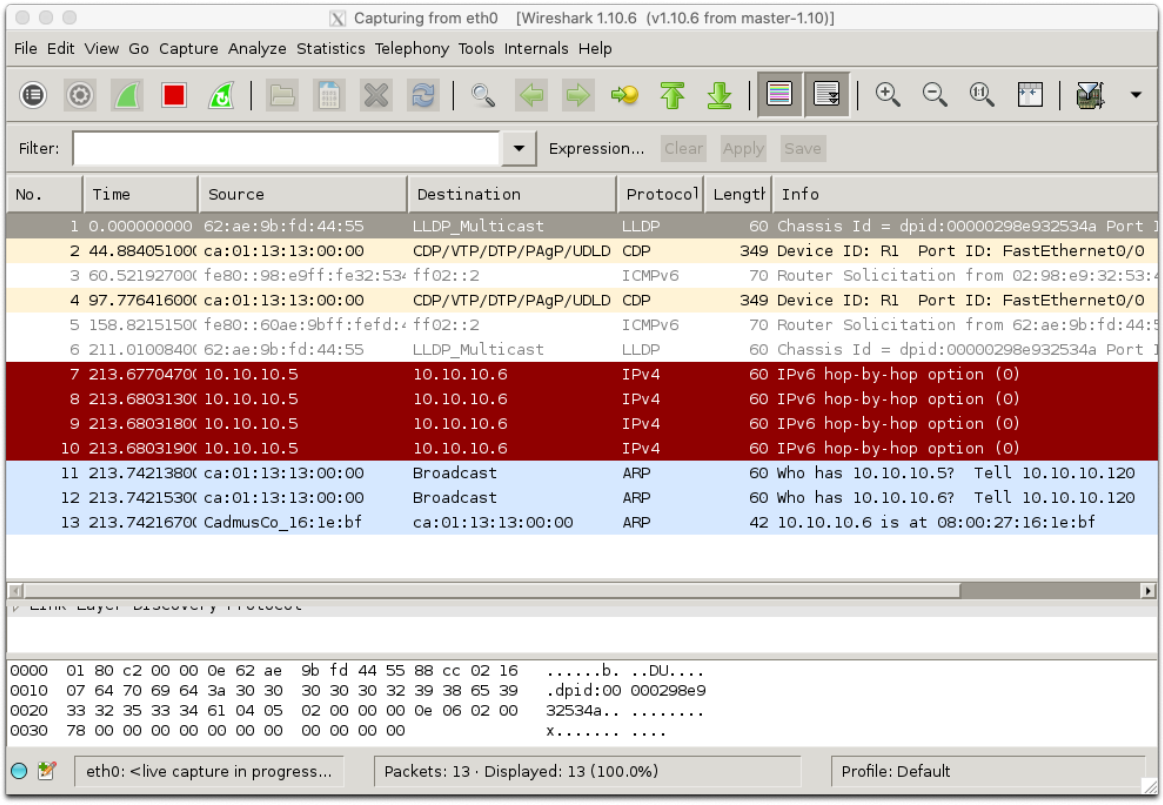


Figura 4.11.- Paquetes capturados en la máquina objetivo del ataque de PVLAN.

4.2.6.- Prueba de detención del ataque

Se procede en este apartado a comprobar si la implementación de la regla en el switch resulta efectiva. Para mostrar este hecho de manera más visual, se ha modificado el código ligeramente respecto del planteamiento inicial. El switch detecta la llegada de un paquete con MAC de destino la del router y éste lo envía al controlador. Aquí se hace una simple comprobación: si la MAC y la IP de destino coinciden con la del router, se deja pasar dicho paquete. En caso contrario, se muestra una alerta avisando del ataque y automáticamente se implementa la regla mencionada en el apartado 4.2.4.1.2

Para extraer la dirección IP de destino, ha de hacerse uso de un módulo proporcionado por Ryu de procesamiento y desempaquetado de tramas IPv4.

El código empleado en el controlador para detener este ataque se muestra con detalle en el código 4.7.

Código 4.7.- Código en aplicación para detectar y mitigar el ataque PVLAN

```
if eth.ethertype == ether_types.ETH_TYPE_IP:
    ip = pkt.get_protocol(ipv4.ipv4)
    if (dst == gateway_mac) and (dstip != gateway_ip):
        self.logger.debug('WARNING: Paquete con MAC destino Puerta de enlace fraudulento')
        self.add_flow_pvlan(datapath,gateway_ip,subnet_mask)
        self.logger.debug('INFO: Nueva regla anadida en el switch satisfactoriamente.')
```

Ha de aclararse que el código anteriormente mostrado no se trata de la versión implementada finalmente, sino con fines únicamente demostrativos. La constante comprobación del tráfico que viaja hacia la puerta de enlace resulta poco eficiente, puesto que aumentaría la latencia del tráfico de red, e incrementaría notablemente la carga computacional tanto del switch como del controlador.

Para la generación del tráfico malicioso en la máquina atacante, se hace uso de nuevo de la herramienta Scapy, por medio de un script en lenguaje Python. En él se detallan la IP tanto de origen como de destino, así como la dirección MAC de la puerta de enlace que se utiliza como punto de apoyo en el ataque. Este script se encuentra detallado en el código 4.8.

Código 4.8.- Script para la generación de los paquetes de ataque a una PVLAN

```
#Importacion de Scapy y todos sus paquetes
from scapy import all as scapy
import sys
def main():
    try:
        packet = scapy.Ether(src="aa:aa:aa:aa:aa:aa", dst="ca:01:13:13:00:00")
        packet = packet/scapy.IP(dst="10.10.10.6",ttl=(1,4))
        scapy.sendp(packet, iface="eth1")
    except Exception as e:
        print(e)
        sys.exit()
if __name__ == "__main__":
    main()
```

Una vez este script se ha implementado correctamente, éste es guardado con el nombre «PVLAN.py» en una carpeta con permisos de ejecución. Para lanzar este ataque, debe ejecutarse como administrador.

```
$ sudo python PVLAN.py
```

Si todo ha salido bien, no se debería observar la llegada de ningún paquete en la interfaz de Wireshark, tal y como se ilustra en la figura 4.12.

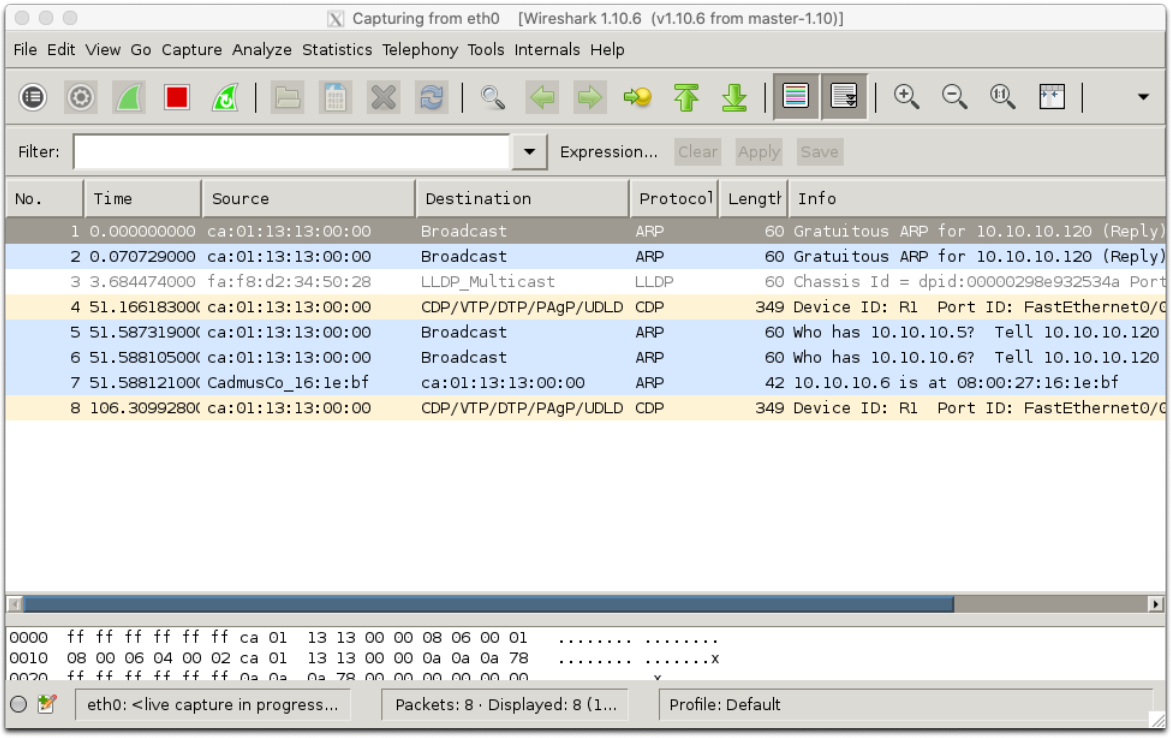


Figura 4.12.- Paquetes capturados en la máquina objetivo del ataque de PVLAN, donde no aparece ninguno perteneciente al ataque.

Así mismo, en la consola de información del controlador, debe aparecer la alerta que ha sido incluida en el código de la aplicación, advirtiendo del ataque y de la correcta implementación de la regla en el switch. Esto se detalla en la figura 4.13

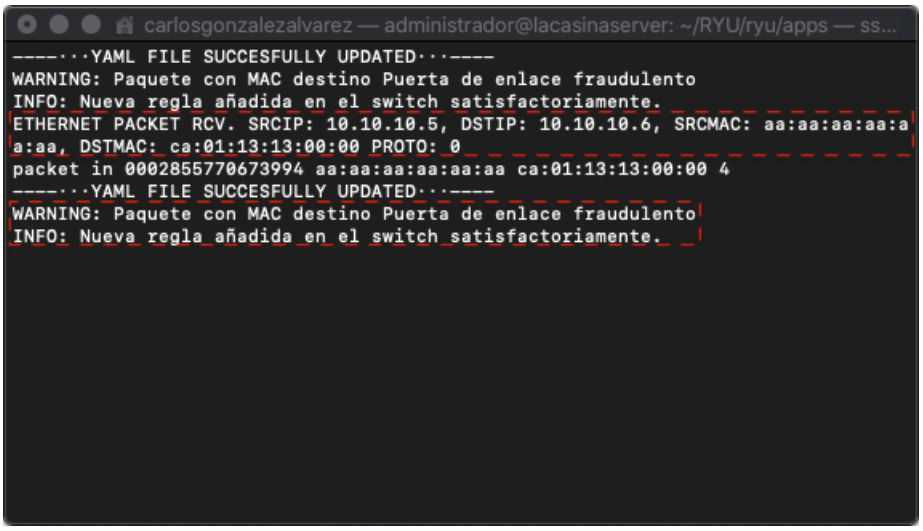


Figura 4.13.- Información mostrada por la consola del controlador, advirtiendo del ataque y de que la regla se ha implementado con éxito.

Se han señalado en la figura 4.13 en color rojo con línea discontinua dos partes:

- La primera, la superior, donde el controlador informa de que se ha recibido un paquete con dirección IP de origen la de la máquina atacante y de destino el objetivo. La MAC de origen es inventada, puesto que es la especificada en el paquete generado con el código 4.8. La MAC de destino es la de la puerta de enlace del escenario.
- En la segunda, la inferior, se observa perfectamente el mensaje de alerta (warning), advirtiendo del ataque, seguido de un mensaje de información indicando que la regla se ha implementado de manera satisfactoria.

Como comprobación final, quedaría ver si, en efecto, la regla está vigente en el dispositivo. En Open vSwitch esto se puede realizar introduciendo el siguiente comando en el dispositivo:

```
# ovs-ofctl dump-flows <iface>
```

«iface» hace referencia a la interfaz del dispositivo en la que se quiere observar todas las reglas y las acciones que llevan asociadas. En este caso particular, esta se corresponde con la etiquetada como **br0**.

```
# ovs-ofctl dump-flows br0
```

Introduciendo este comando, se muestran todas las reglas del switch. En la figura 4.14 se representa la información de las tablas de flujo del switch. Se ha subrayado en azul claro la reglas de interés y la acción asociada (*drop*, en este caso, que es descartar).

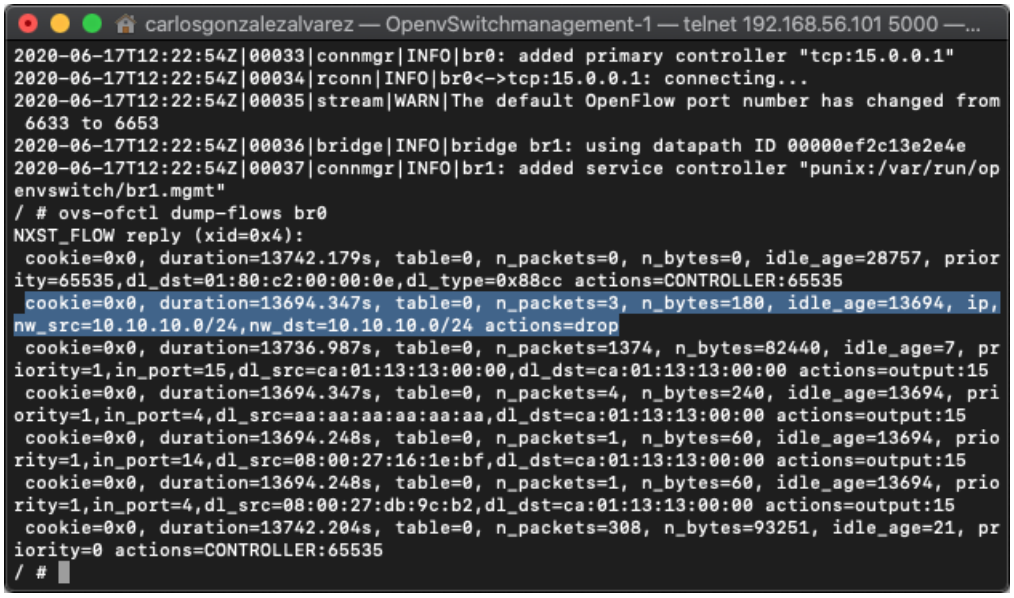


Figura 4.14.- Tabla de flujos del controlador, en la que se observa la regla de restricción de comunicación entre dispositivos de la misma red, con el fin de mitigar el ataque de PVLANs.

4.3.- Ataque de doble tagging

Las VLAN (*virtual local area network*) son segmentos lógicos dentro de una misma red física. Estas divisiones de una red en varias más pequeñas permiten reducir el dominio de difusión y ayudan a la administración de la red. Estos segmentos se administran por medio de un switch que permita tal acción para así albergar diferentes redes VLAN.

La comunicación entre equipos de una misma VLAN está permitida, no siendo así entre equipos que pertenezcan a segmentos de redes VLAN distintas. Este aspecto es muy importante tenerlo en cuenta sobre todo en lo que respecta a temas de seguridad.

Una misma red virtual puede pertenecer a múltiples switches, estando por tanto la conectividad asegurada. Esto es posible mediante la conexión de los switches entre sí y definiendo lo que se denominan «puertos troncales» (*trunk ports*), que permiten el paso de información de múltiples VLAN entre switches. Cada red virtual tiene asignada un identificador único, que consiste en un número comprendido entre 1 y 4096.

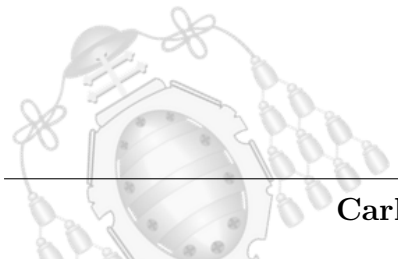
A pesar de que la conectividad entre equipos de distintas VLAN no es posible, existen métodos que permiten saltarse esta restricción. Uno de ellos es el conocido como ataque de doble etiqueta, o *double tagging* en inglés. Este ataque aprovecha el funcionamiento de algunos switches a la hora de tratar el etiquetado de VLAN en las tramas Ethernet en los enlaces troncales.

4.3.1.- Escenario del ataque

Se presupone un escenario de red como el mostrado en la figura 4.15. Cuenta con dos VLAN:

- **VLAN 20**: Es la nativa y pertenece al enlace troncal entre los dos dispositivos conmutadores de red. A esta VLAN pertenece también el atacante, con dirección IP 10.10.10.1 en una red 10.10.10.0/29
- **VLAN 30**: Es la objetivo de ataque en este caso, cuya dirección de red es la 10.10.10.8/29 y el objetivo una dirección IP 10.10.10.10.

El objetivo de este ataque es que el usuario malicioso pueda enviar tráfico desde la VLAN 20 hasta la 30.



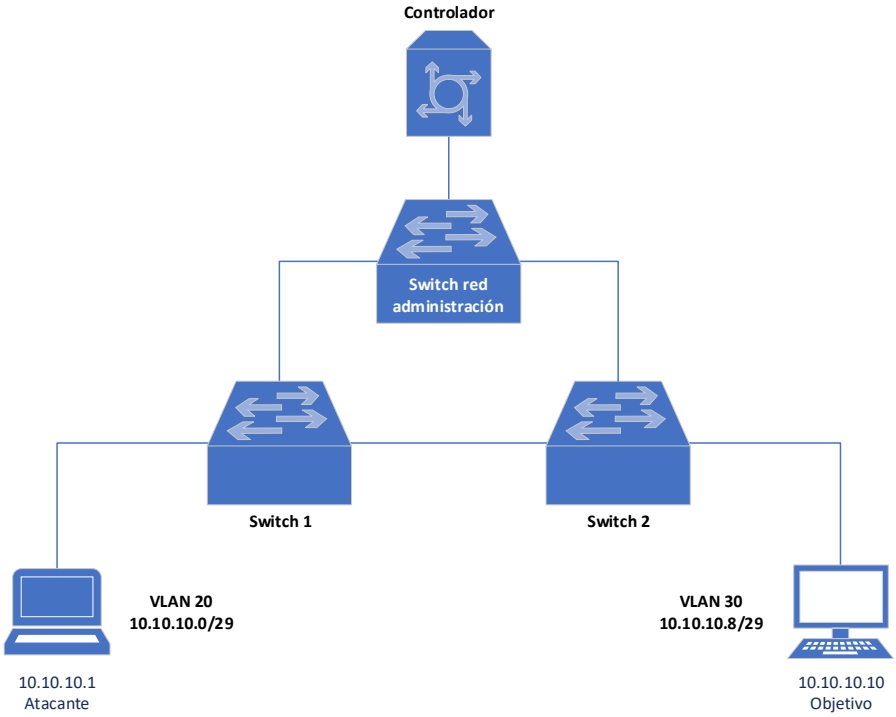


Figura 4.15.- Escenario empleado para el ataque de doble etiquetado.

4.3.2.- Procedimiento del ataque

La VLAN nativa se utiliza en aquellos puertos del switch configurados como troncales. Es utilizada de manera implícita para todo el tráfico que viaje sin etiqueta a través de estos puertos. Por defecto en la mayoría de switches, la VLAN nativa tiene como identificador el 1.

Para realizar el ataque de doble etiqueta, el atacante se conecta a un puerto que pertenece a la VLAN nativa, en el caso de este escenario la 20. Éste envía una trama con dos etiquetas: la más externa, la de la nativa y otra más interna con el identificador de la red virtual objetivo de ataque, en este caso la 30. Esta trama llega al switch número 1, el cual quita la etiqueta más «externa» que es la de la nativa y la envía por el enlace troncal.

La trama que viaja por el enlace entre los dos switches contiene entonces una única etiqueta, la de la VLAN 30. Cuando ésta llega al segundo de los switches, al observar la etiqueta, envía la trama correspondiente al equipo de la VLAN 30.

4.3.3.- Mitigación del ataque en una red tradicional

Existen varias formas de evitar este tipo de ataques entre redes VLAN distintas. El primero, y más evidente, es evitar que un usuario cualquiera pueda conectarse, de forma directa, a un puerto troncal del switch. Sin embargo, existen ataques que permiten cambiar un puerto normal y convertirlo en troncal, como por ejemplo, utilizando el protocolo DTP (*dynamic trunk port*).

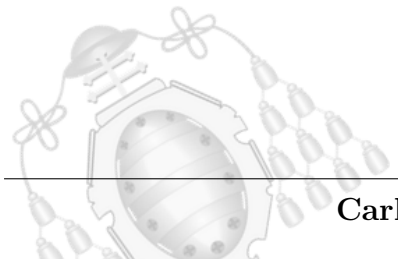
Otro enfoque pasaría por cambiar el comportamiento del switch a la hora de procesar el tráfico proveniente de una VLAN nativa. Eso supone etiquetar todo el tráfico que viaje por enlaces troncales. Se trata de una solución más elegante puesto que, independientemente de si un usuario puede acceder o no a través de un puerto troncal, este ataque no sería efectivo. Para adoptar la segunda solución, en el plano de las redes tradicionales, depende de si el equipamiento del cual se dispone permite realizar estas acciones sobre el tráfico. Cada fabricante puede adoptar distintas estrategias de etiquetado en este tráfico.

4.3.4.- Mitigación del ataque en un entorno SDN

En las redes definidas por software, el controlador puede actuar sobre los paquetes modificando sus campos, añadiendo o eliminando etiquetas existentes. Así mismo, estas acciones pueden recaer sobre los switches de la red, gracias a las acciones incluidas en las tablas de flujo correspondientes.

4.3.4.1.- Creación de la aplicación

Para trabajar en el plano de VLANs con Ryu es necesario implementar toda su lógica en la aplicación a desarrollar. Existen diferentes aplicaciones de partida en la comunidad de desarrolladores del controlador que proveen de esta capacidad a los switches, y que van a ser la base de partida para detener ataques de doble etiquetado.



La configuración de cada dispositivo debe incluirse en la aplicación, bien en el propio código -práctica no recomendable- o a través de un fichero de configuración externo en el que, para cada dispositivo de la red, se provea la siguiente información:

- VLANs asociados a cada puerto.
- Qué puertos son troncales.
- Qué puertos son de acceso.

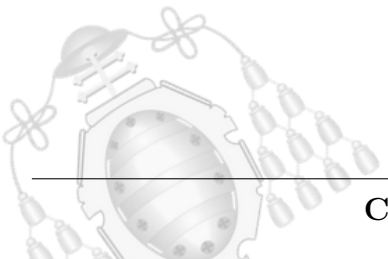
La aplicación que viene por defecto, y que contiene toda la lógica de las VLANs para dispositivos que utilicen como controlador Ryu, se adjunta en el Anexo VII.

En síntesis, lo que realiza es una inspección de los paquetes que llegan al controlador. El administrador de la red debe indicar, para cada puerto, las VLANs correspondientes y su modo (acceso o troncal). Una vez hecho esto, el controlador comprueba si los paquetes que le llegan contienen algún tipo de etiqueta VLAN. Si cumplen este criterio, se analiza qué puertos tienen asociada esa VLAN, gracias a la información proporcionada previamente:

- Si esos puertos son de acceso, se comprueba si tras ese puerto se encuentra el destinatario de dicho paquete, gracias a la asociación MAC-puerto que realiza el controlador en todo momento. De ser así, se envía el paquete sin etiquetar a ese destino.
- Si esos puertos son troncales, se envía el paquete sin modificar a través del puerto o puertos troncales correspondientes.

Por defecto, no se comprueba si el paquete que es enviado por un enlace troncal está etiquetado con la VLAN nativa de este enlace. Es por ello que son posibles estos ataques de doble etiquetado.

Para corregir esto, se debe cambiar el comportamiento del controlador cuando un paquete llega procedente de una VLAN con el mismo identificador que la VLAN nativa de un enlace troncal.



4.3.4.1.1.- Cambio en el procesamiento de los paquetes

Para evitar los ataques de doble etiquetado, se deben efectuar cambios a nivel de aplicación que modifiquen la forma en que el controlador los procesa una vez llegan. Este cambio es el siguiente: todo tráfico recibido en un puerto troncal, sin etiquetar, debe ser etiquetado con la VLAN nativa antes de atravesar el troncal. En caso de que ya viniese etiquetado con la nativa, no se hace nada. Para el resto de casos, el funcionamiento sigue siendo el que viene marcado por defecto.

4.3.4.1.2.- Implementación de los cambios

Los cambios se realizan dentro de la función de procesamiento de paquetes en el controlador, **packet_in**. Se comprueba si el paquete contiene cabecera VLAN presente, a través de la variable **vlan_header_present**. De ser así, el siguiente paso es determinar si los puertos de salida de dicho paquete son troncales o no. Esto se realiza con la variable **out_port_trunk** que contiene un vector con todos los puertos candidatos de salida y que son troncales.

Una vez realizada esta comprobación, el siguiente paso es comprobar si el paquete tiene como etiqueta más externa la de la VLAN nativa del puerto troncal de destino. Esto se realiza gracias a la información proporcionada por el administrador de la red, en las variables de **port_vlan**, **access** y **trunk**.

Para realizar esta comprobación, se crea una función, **check_out_port_trunk**, que recibe los siguientes parámetros:

- **dpid**: Número de identificación único del switch, para realizar las comprobaciones de los puertos en las listas proporcionadas por el administrador de la red.
- **out_port_trunk**: Lista que contiene todos los puertos troncales candidatos para la salida del paquete.
- **src_vlan**: Número identificativo de la VLAN presente en el paquete recibido. En caso de contener múltiples etiquetas, siempre se obtiene el de la etiqueta más externa.

La función itera sobre la lista de puertos troncales en busca de si alguna de sus VLAN coincide con la presente en el paquete. De ser así, se devuelve una tupla con los pares [True, null]. En caso contrario, se devuelve una tupla con los pares [False, vlan_id] donde **vlan_id** es el valor de la VLAN nativa en este caso. Esta función se representa en el código 4.9

Código 4.9.- Función que comprueba la VLAN nativa de destino de un puerto troncal

```
def check_out_port_trunk(self,dpid,out_port_trunk,src_vlan):
    checker = [False,""]
        for port in out_port_trunk:
            if src_vlan in port_vlan[dpid][port]:
            checker = [True,port_vlan[dpid][port]]
            return checker
        checker[1]=port_vlan[dpid][port]
        return checker
```

En el caso que la tupla sea [True, null], el paquete se procesa de manera normal, sin modificar ninguna de sus etiquetas. Se suprime la acción de eliminar la primera etiqueta, como se realizaba hasta el momento.

Si el valor devuelto por la función es [False, vlan_id] se debe añadir al paquete una etiqueta con el valor presente en la variable **vlan_id**. Esto es posible gracias a las instrucciones **OFPActionPushVlan** y **OFPActionSetField** de Ryu.

En la primera instrucción se pasa como argumento el tipo de paquete, que para este caso es de tipo 802.1Q, cuyo identificador en hexadecimal es **0x8001** y en decimal es **33024**.

Para la segunda instrucción, se indica el identificador de la VLAN que se quiere introducir en el paquete. Este conjunto de acciones se guardan en un vector, por medio de las funciones append.

```
actions = []
actions.append(parser.OFPActionPushVlan(33024))
actions.append(parser.OFPActionSetField(vlan_vid=src_vlan))
```

A estas acciones, se le suman también aquellas que envían el paquete por todos los puertos candidatos troncales, incluidos en la variable **out_port_trunk**.

```
for port in out_port_trunk:
   actions.append(parser.OFPActionOutput(port))
```

4.3.5.- Prueba de la realización del ataque

Al igual que se comentaba en el apartado 4.2.5, se cuenta con un programa de generación de paquetes en la máquina atacante y un software para capturar tráfico en el equipo objetivo, con el fin de comprobar si el tráfico llega y el ataque ha resultado exitoso.

Para la generación del tráfico malicioso en la máquina atacante, se hace uso de nuevo de la herramienta Scapy, por medio de un script en lenguaje Python. En el se detallan la IP tanto de origen como de destino, así como la dirección MAC del destino. Debe incluirse las dos etiquetas, representadas por la clase Dot1Q de Scapy. Este script se encuentra detallado en el código 4.10, y es análogo al utilizado en anteriores ataques, véase el código 4.8. Para este caso particular, el paquete que se envía es un «ping», perteneciente al protocolo ICMP.

Código 4.10.- Script para la generación de los paquetes de ataque de doble etiqueta

```
from scapy import all as scapy
import sys
def main():
    try:
        packet = scapy.Ether(dst='08:00:27:16:1e:bf', src='08:00:27:db:9c:b2')
        packet=packet/scapy.Dot1Q(vlan=20)/scapy.Dot1Q(vlan=30)
        packet=packet/scapy.IP(dst='10.10.10.10', src='10.10.10.1')/scapy.ICMP()
        scapy.sendp(packet, iface="eth1")
    except Exception as e:
        print(e)
        sys.exit()
if __name__ == "__main__":
    main()
```

El primer switch recibe el paquete generado a través del puerto troncal 4, que da acceso al atacante. Siguiendo el procedimiento que tienen los switches por defecto, elimina la primera etiqueta y la envía por el enlace troncal, cuya VLAN nativa es la misma que la del atacante. Dicho paquete es recibido por el puerto 10 del segundo switch con únicamente una etiqueta, la de la VLAN 30, cuyo puerto asociado es el número 2. La trama llega finalmente al destino del ataque.

Para poder tener la trazabilidad del paquete en todo momento, se han habilitado mensajes de depuración en el controlador que informa en todo momento del puerto de llegada del paquete así como de la etiqueta de VLAN que tiene asociada más externa. Al ejecutar el script, aparecen los mensajes que pueden observarse en la figura 4.16.

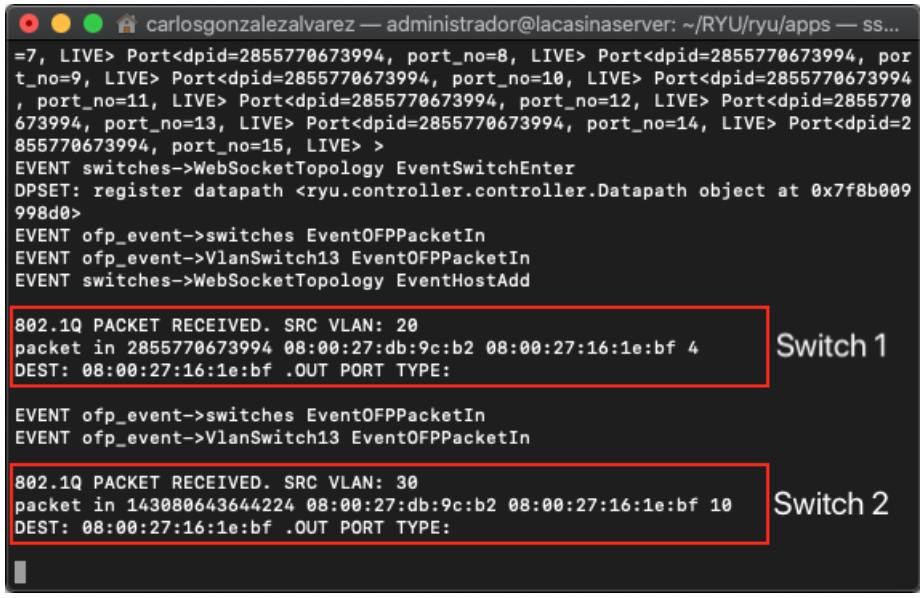


Figura 4.16.- Mensajes mostrados por el controlador en el ataque de doble tagging, en la que se observa la etiqueta más externa del paquete en la llegada a cada dispositivo.

En la figura 4.16 se han señalado dos partes. En la superior se muestra el mensaje proveniente del primer dispositivo. Señala que la etiqueta de la VLAN más externa es la número 20. Tal y como se establece en el funcionamiento del switch, esta etiqueta externa es eliminada y el paquete se envía por el puerto troncal.

El recuadro inferior, muestra el mensaje proveniente del segundo switch. El paquete es recibido por el puerto 10 y contiene la etiqueta de la VLAN número 30, la segunda

de las que contenía. En consecuencia, el switch interpreta que ese paquete es para el ordenador víctima, reenviándolo por su correspondiente puerto.

Puede observarse la llegada del paquete mediante el programa Wireshark en la máquina objetivo, como se observa en la figura 4.17.

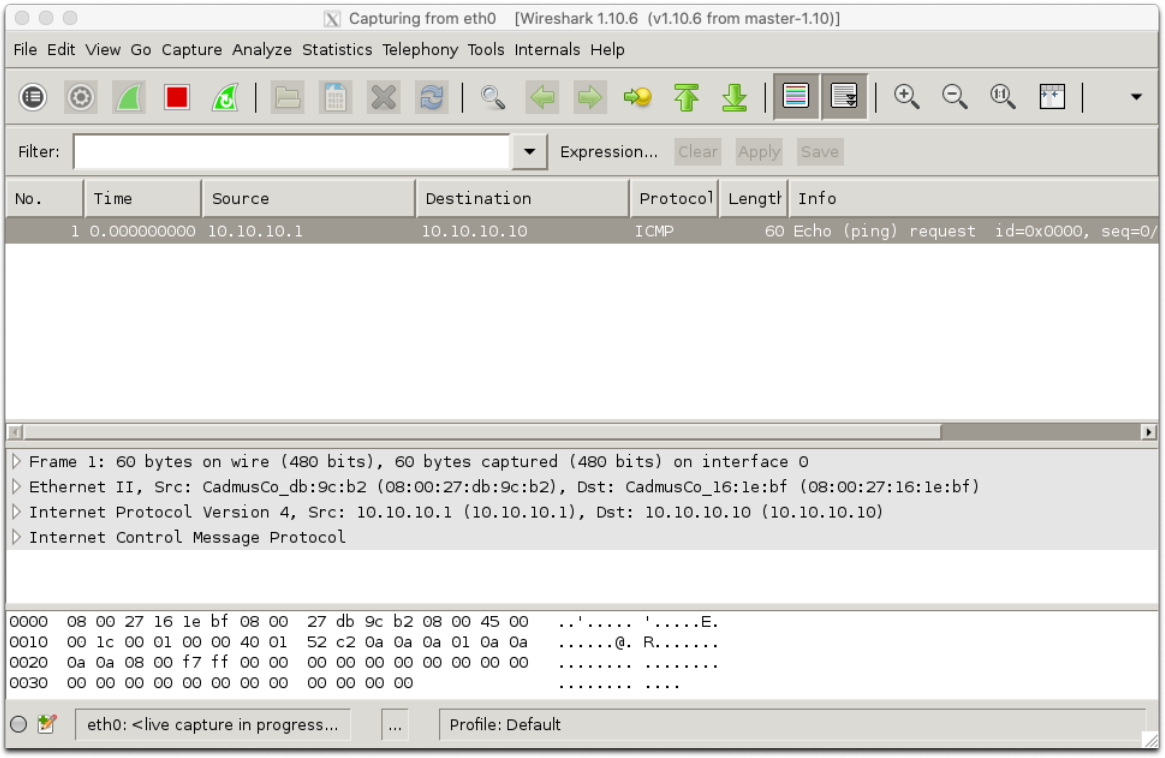


Figura 4.17.- Paquete capturado por Wireshark en la máquina receptora del ataque de doble tagging.

Mediante la doble etiqueta, el tráfico llega al destino tal y como estaba previsto. Esto abre la puerta a ataques unidireccionales hacia el objetivo, desde denegación del servicio hasta el envío de malware vía UDP.

4.3.6.- Prueba de la detención del ataque

Ahora se procede a ejecutar el mismo tipo de ataque, pero con las modificaciones en el funcionamiento de la red efectuadas y explicadas en el apartado 4.3.4.1.

Se mantiene el mismo escenario, lo único que cambia es la aplicación que se ejecuta en el controlador, cuyo objetivo es evitar este tipo de ataques. Se ejecuta el mismo

script que en el apartado anterior, donde se envía un paquete con doble etiqueta. Al haber modificado el comportamiento de los switches, para ambos debe llegar el paquete con la misma etiqueta externa, en este caso, con la de la VLAN 20. El resultado en el controlador, tras la ejecución del ataque, se muestra en la figura 4.18.

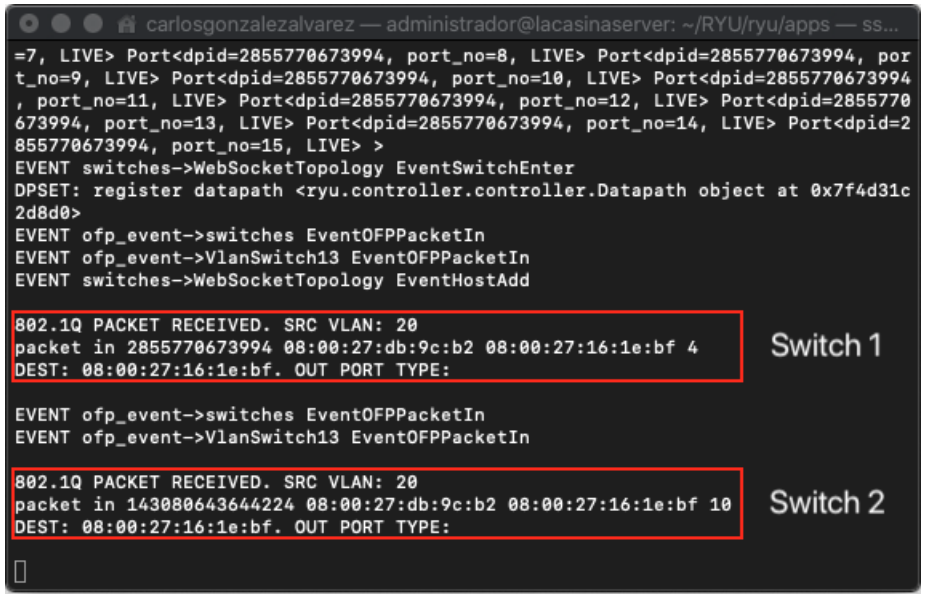


Figura 4.18.- Mensajes mostrados por el controlador en el ataque de doble tagging, en la que se observa la etiqueta más externa del paquete en la llegada a cada dispositivo.

En la figura 4.18 se han señalado en rojo, de nuevo, dos partes. En la superior se muestra el mensaje proveniente del primer dispositivo. Indica que la etiqueta de la VLAN más externa es la número 20. Esta etiqueta, al corresponderse con la de la VLAN nativa del enlace troncal, se mantiene y es enviada por los troncales diferentes al recibido. A su llegada al segundo switch, se puede comprobar este efecto, puesto que la etiqueta más externa sigue siendo la 20, como puede verse en la parte inferior señalada en rojo.

Como era previsible, en la máquina objetivo este paquete no es recibido, tal y como se observa en la figura 4.19.

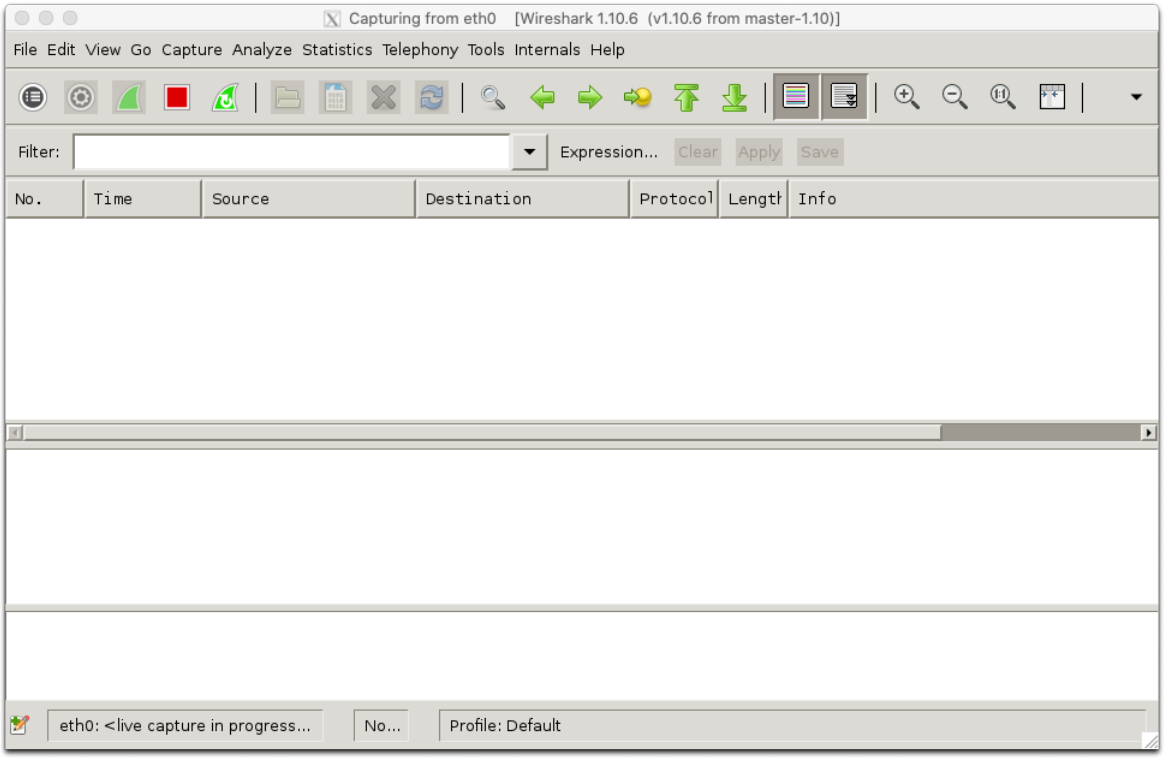


Figura 4.19.- Captura de Wireshark en la máquina objetivo, en la que no se observa la llegada de ningún paquete.

Se demuestra, con ello, la eficacia de la aplicación para la mitigación de estos ataques de doble etiqueta en el entorno de las VLAN.

4.3.7.- Mejoras

Al igual que se ha realizado con anteriores aplicaciones, se han aplicado algunas mejoras orientadas a la facilidad de uso por parte de los usuarios. Otras se comentan como posible ampliación de cara a futuro, centradas en la mejora del rendimiento del controlador en particular y la red en general.

4.3.7.1.- Fichero de configuración

Todas las configuraciones de las VLAN por dispositivo se llevan a un fichero externo, en formato YAML. Para cada switch, se indican los puertos de acceso, los troncales y las VLAN que tienen asociadas. Cuenta con la siguiente estructura:

```
port_vlan = {
      2855770673994: {
         10:[20],
         4:[20]
         },
      143080643644224: {
         2:[30],
         10:[20]
         }
      }

access= { 2855770673994:[],
           143080643644224:[2]
           }

trunk = { 2855770673994:[4,10],
          143080643644224:[10]
         }
```

Con ello, se evitan técnicas de *hardcoding*, nada recomendables en el mundo de la programación.

4.4.- Protocolos de enrutamiento y redundancia de primer salto

Numerosos ataques tienen como objetivo modificar el comportamiento de los elementos de red, entre ellos, los routers. El atacante, haciendo uso de protocolos de red específicos, puede conseguir hacerse pasar por una puerta de enlace y capturar todo el tráfico. Entre estos protocolos están los de enrutamiento y los de redundancia de primer salto.

Para que un entorno de red LAN tradicional sea seguro debe hacerse una correcta configuración de estos protocolos. En el ámbito de estudio de este proyecto, se pretende aplicar técnicas de SDN para mitigar los problemas de seguridad que una mala configuración de estos protocolos puede provocar.

4.4.1.- Protocolos de enrutamiento

Los protocolos de enrutamiento tienen como objetivo la creación y el mantenimiento de las tablas de rutas. Éstas contienen todas las redes conocidas, así como las interfaces asociadas a esas redes. Los routers utilizan estos protocolos para obtener información bien de otros routers, de la configuración de sus interfaces o mediante la inclusión manual de rutas por parte de los administradores de la red.

Un envenenamiento de las tablas de rutas de un dispositivo consiste en el envío de información de enrutamiento fraudulenta, con el fin de modificar estas tablas, estableciendo nuevos itinerarios para el tráfico hacia dispositivos no confiables. Con ello, se modifica el comportamiento de la red y el tráfico por el que viajan los paquetes, pudiendo ser interceptados por terceros y prometiendo con ello la seguridad e integridad de las comunicaciones.

Dos de los protocolos más utilizados en redes son Open Shortest Path First (OSPF) y Enhanced Interior Gateway Routing Protocol (EIGRP). Ambos proporcionan mecanismos de autenticación y seguridad. Sin embargo, esto requiere una configuración minuciosa de los dispositivos que empleen dichos protocolos.

4.4.2.- Protocolos de redundancia de primer salto

Cuando una red LAN requiere de redundancia, pueden configurarse múltiples routers como puertas de enlace. Estos protocolos permiten ver dichas puertas de enlace como una única, de forma que el usuario no necesita realizar ningún tipo de configuración adicional.

Esta circunstancia de varias puertas de enlace puede ser aprovechada para la realización de ataques o de captación de tráfico. Un usuario puede hacerse pasar como puerta de enlace activa y encaminar, de manera fraudulenta, todo el tráfico legítimo hacia sí mismo.

Existen varios protocolos de redundancia de primer salto, como por ejemplo el *Hot Standby Routing Protocol* (HSRP) de Cisco, donde un router es el activo y el resto se

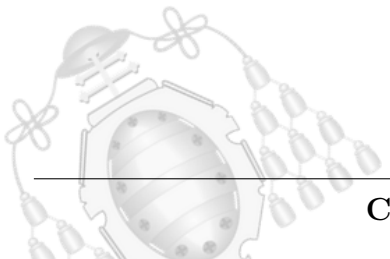
mantienen como **backup**, en caso de que el principal falle. Proporciona mecanismos de autenticación tanto por texto plano como por MD5 para evitar problemas de seguridad, lo cual requiere de una configuración adicional por parte del administrador de red, puesto que no viene activado por defecto.

4.4.3.- Escenario del ataque

El escenario objeto de trabajo para este apartado está constituido por dos redes: La 10.10.10.0/24 y la 120.10.10.0/24. Ambas cuentan con sus puertas de enlace, 10.10.10.120 y 120.10.10.1 respectivamente.

La red 10.10.10.0/24 cuenta con dos usuarios, el atacante y la víctima del ataque. El objetivo es interceptar el tráfico que tenga como destino la red 120.10.10.0/24.

El router R1 se encuentra conectado al puerto 15 del switch, el atacante al puerto 4 y el objetivo al 13.



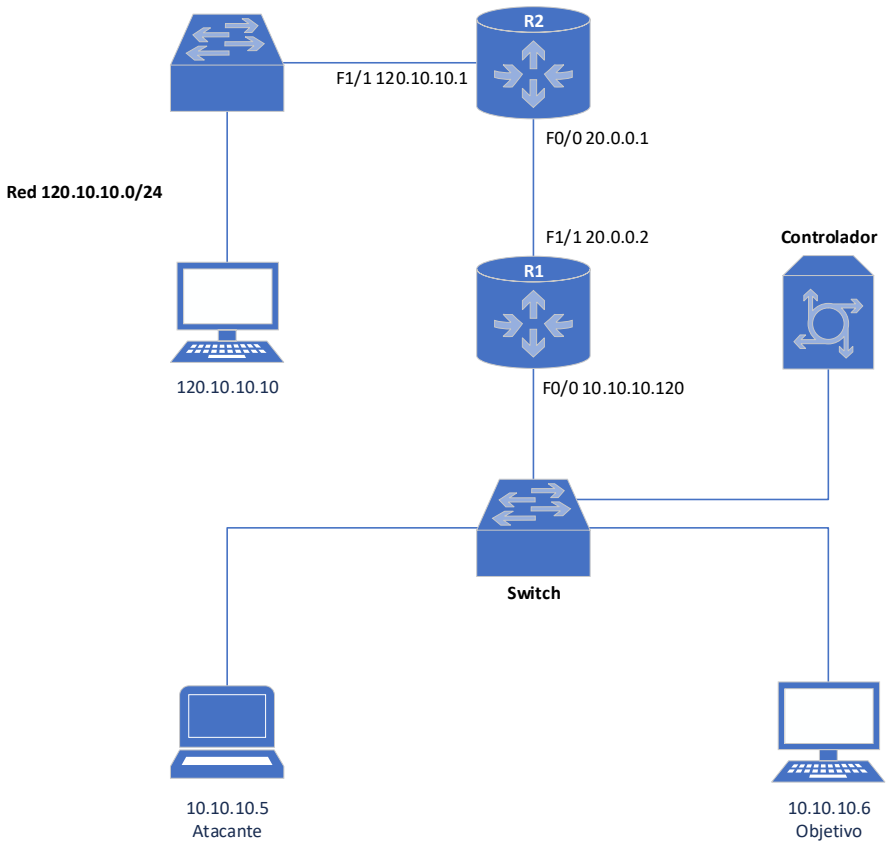


Figura 4.20.- Escenario modelo para simulación de los ataques de envenenamiento de tablas de rutas.

4.4.4.- Procedimiento del ataque

Dependiendo de la familia de protocolos, el procedimiento varía ligeramente aunque el objetivo, en esencia es el mismo: captar todo el tráfico legítimo.

En el caso de los protocolos de enrutamiento, el usuario malicioso se hace pasar por un router, enviando información de enrutamiento con parámetros modificados. El objetivo es que los routers «aprendan» nuevas rutas hacia redes conocidas, siendo el paso a través del atacante. El atacante puede obtener información en la red capturando el tráfico generado por los routers y así conocer, por ejemplo, el número de saltos que existen hacia una determinada red y poder «ofrecer» esa misma red a un coste menor, como podría ser el caso del protocolo RIP.

Para protocolos de redundancia de primer salto, el usuario se hace pasar por la puerta de enlace activa. En el caso de HSRP, por ejemplo, existen multitud de herramientas que aprovechan las vulnerabilidades que ofrece este protocolo si no se autentican los mensajes, siendo Yersinia una alternativa a tal efecto. Con un simple comando el usuario puede hacerse pasar por el router activo y ejercer como puerta de enlace legítima en una red con redundancia de primer salto.

4.4.5.- Mitigación del ataque en redes convencionales

Dependiendo del protocolo y sus opciones de configuración, las formas de proteger de este tipo de ataques varían. Para este apartado se desglosan los siguientes protocolos: RIP, OSPF y EIGRP para los protocolos de enrutamiento y HSRP para los protocolos de redundancia de primer salto.

4.4.5.1.- Protocolos de enrutamiento

En los siguientes apartados se comentan las medidas que se deben implementar para mitigar los ataques en los principales protocolos de enrutamiento en redes convencionales.

4.4.5.1.1.- OSPF

Es un protocolo de puerta de enlace interna, utilizado para distribuir información de rutas entre sistemas autónomos. Utiliza el algoritmo de Dijkstra para calcular el camino más corto entre dos nodos.

Existen dos maneras de proteger las comunicaciones por mensajes del protocolo OSPF: mediante la autenticación de los mensajes o la declaración de aquellas interfaces por las que no se desean «aprender» nuevas rutas como «pasivas».

En el caso primero, el de la autenticación de mensajes, OSPF permite dos formas distintas de configurarla:

- **Autenticación por texto plano**: Conocida también como «tipo 1», utiliza contraseñas sin cifrar en las cabeceras de los mensajes.
- **Autenticación MD5**: O de «tipo 2» y cifra las contraseñas mediante el algoritmo MD5.

En caso de no utilizar autenticación, o como método adicional a éste, se emplea el uso de lo que se denominan «interfaces pasivas». Esto es, indicar en dichos puertos del router, que no se van a formar relaciones de adyacencia con los routers vecinos, lo que posibilitaría el intercambio de rutas entre ellos.

4.4.5.1.2.- EIGRP

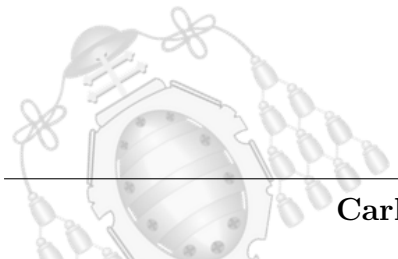
Es un protocolo de encaminamiento basado en vector-distancia propietario de Cisco. Aúna también algunas de las características de otros protocolos de estado enlace, como OSPF.

Permite la autenticación de rutas mediante MD5 en cada mensaje de actualización. Cada clave MD5 tiene su propio identificador, almacenado en cada router de manera local. La combinación de este identificador único, ligado a la interfaz asociada a ese mensaje, previene la introducción de mensajes falsos o no autorizados de fuentes no fiables.

Dichas claves han de ser introducidas por el propio administrador de la red, puesto que no se encuentran activadas de inicio al utilizar este protocolo.

4.4.5.1.3.- RIP

Las siglas RIP se corresponden con *Routing Information Protocol*. Es de los primeros protocolos de encaminamiento existentes. Utiliza algoritmos de vector-distancia, basados en el número de saltos entre origen y destino para el cálculo de las rutas. A fecha actual, existen dos versiones, la 1 (RIPv1) y la 2 (RIPv2).



RIPv1 no ofrece ningún tipo de autenticación, lo que hace más vulnerable a este protocolo a ataques. La versión 2 introduce autenticación, mediante los anteriormente mencionados MD5 o texto plano.

4.4.5.2.- Protocolos de redundancia de primer salto

En cuanto a los protocolos de redundancia de primer salto, en siguientes apartados se comentan las medidas que se deben implementar para mitigar los ataques en HSRP, el protocolo objeto de estudio en este proyecto.

4.4.5.2.1.- HSRP

Es un protocolo propietario de Cisco y se utiliza para la redundancia de puertas de enlace. Una es la activa, mientras que el resto permanece como respaldo en caso de fallo de la principal. Un usuario malintencionado, puede aprovechar este protocolo para hacerse pasar como puerta de enlace más prioritaria y recibir todo el tráfico, pudiendo realizar ataques *Man in the Middle*.

Al igual que OSPF, HSRP ofrece autenticación de los mensajes mediante texto plano y el algoritmo MD5. Esto debe ser habilitado por el administrador de la red, puesto que la opción que se establece por defecto es la de mensajes sin ningún tipo de autenticación.

Otra posibilidad de mitigar la llegada de mensajes no deseados, es mediante el establecimiento de listas de control de acceso, especificando el origen legítimo de los mensajes HSRP.

4.4.6.- Mitigación del ataque en el plano de las SDN

El problema del envenenamiento de rutas puede solucionarse aplicando técnicas de SDN sobre los switches o realizando determinadas acciones en el controlador. De manera general, bastaría con permitir únicamente el tráfico de estos protocolos en los puertos legítimos, es decir, los que tengan conectados routers. Para el resto de puertos

a los cuales no esté conectado ningún router, este tráfico al ser recibido en el switch se descartaría automáticamente. Sin embargo, han de tenerse en cuenta ciertos aspectos de configuración que pueden ser propios a cada sistema, como por ejemplo, la existencia de puertos troncales o las distintas topologías de red existentes.

4.4.6.1.- Creación de la aplicación

Siguiendo las directrices de los anteriores ataques, se establecen en los switches una serie de reglas de flujo, acorde con las exigencias que quiera establecer el administrador de la red. Si bien la inclusión de reglas es una opción válida, existen otros enfoque igualmente válidos para mitigar estos ataques en el plano de las SDN.

- Al igual que se ha realizado en anteriores ataques, se establecen criterios de flujo de tráfico en los switches. Aquellos que satisfacen la regla de pertenencia a un determinado protocolo de enrutamiento, son automáticamente descartados.
- Debido al disminuido volumen de tráfico de estos protocolos que circula por la red en condiciones normales, todo paquete o trama perteneciente a dichos protocolos de enrutamiento se redirige al controlador y es allí donde se determina si su procedencia es legítima o no.

Teniendo en cuenta la baja carga computacional que supondría para el controlador el reenvío de este tráfico y su procesamiento, así como por tratarse de un enfoque mucho más flexible y personalizable, se opta por la segunda opción.

La discriminación de paquetes en base al protocolo de enrutamiento al que pertenecen no siempre se trata de una operación trivial. El caso particular de EIGRP y OSPF, pertenecientes a la capa de red, pueden ser fácilmente localizados gracias al campo **ip_proto** contenido en la cabecera de mensajes IP. Este criterio es uno de los múltiples que OpenFlow permite utilizar para la creación de las reglas en las tablas de flujo.

Común a todos estos protocolos es la dirección de IP de destino de sus paquetes. Tal y como especifica el estándar RFC 3171, el rango de direcciones comprendido entre la 224.0.0.0. y la 239.255.255.255 está destinado a multicast, es decir, a la difusión de

contenido para una serie de receptores interesados. Cualquier paquete que llegue a un switch en una red SDN con dirección IP de destino multicast, no va a encontrar, a menos que se haya especificado previamente algún tipo de regla, coincidencias en la tabla del switch. Esto supone que se aplica la opción por defecto establecida por el administrador de red. En el caso de este trabajo, esta opción es la del envío al controlador. Es por ello que, de partida, está asegurado que todo paquete de protocolos de enrutamiento es reenviado al controlador.

En cuanto a la aplicación, todas las operaciones de comprobación de los paquetes de llegada en el controlador se realizan en la función predefinida **_packet_in_handler**.

Por medio del uso de funciones definidas en la biblioteca de Ryu, es posible extraer el tipo de paquete que llega al controlador, así como el id del protocolo asociado, en caso de tratarse de una trama IP. Cada cabecera de un paquete IP contiene un campo de 8 bits de longitud que se corresponden con el identificador del protocolo que tienen asociado, como puede observarse en la figura 4.21. Estos valores, pueden consultarse en la literatura o cualquier fuente de información sobre el protocolo. En este caso concreto, interesa conocer los valores que tienen asociados los protocolos EIGRP y OSPF en dicho campo: EIGRP tiene el valor 88 y OSPF el 89.

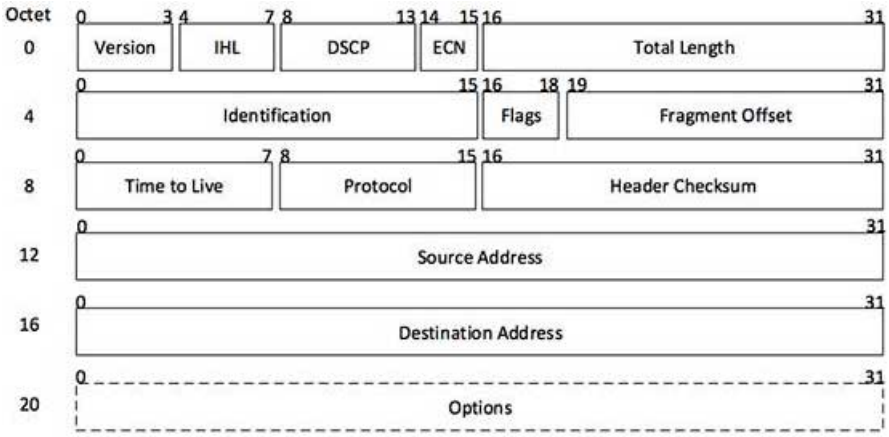


Figura 4.21.- Cabecera de un paquete IP.

Para el caso de RIP, al tratarse un protocolo de la capa de aplicación, debe utilizarse otra estrategia para su detección bien en las tablas de flujo de los switches o en su

procesamiento en el controlador. En este caso, el puerto UDP que tenga asociado, característico para este protocolo, que es el 520.

En cuanto al tráfico HSRP, el procedimiento es análogo al indicado para RIP, con algún añadido. La dirección de destino de los mensajes HSRP es la multicast 224.0.0.2, y el puerto tanto de origen como de destino es el 1985.

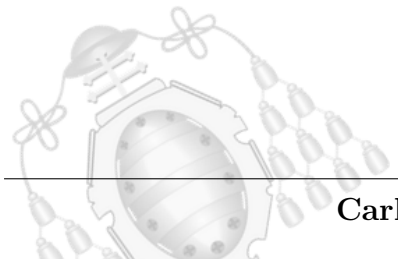
Para todo paquete entrante, se examina si cumple las condiciones anteriormente expuestas. De ser así, el siguiente paso es comprobar si el puerto de ingreso en el switch está habilitado para recibir tráfico perteneciente a los protocolos de enrutamiento. En ese caso, se procesa normalmente siguiendo las directrices del controlador. Si dicho puerto no está habilitado para recibir tal tráfico, se detiene el proceso de tratamiento del paquete mediante una simple instrucción de «return».

Los puertos del switch habilitados para recibir tráfico de protocolos de enrutamiento, se definen en una variable dentro del código fuente de la aplicación, denominada **Ports_enabled**. Consiste en un vector que contiene el número identificativo de cada puerto por el que se quiera recibir dicho tráfico.

```
#Para recibir paquetes de protocolos de enrutamiento por los puertos 1, 3 y 5
Ports_enabled = [1, 3, 5]
```

A modo de depuración, y para que se muestren mensajes de alerta en el controlador, se añaden una serie de advertencias cuando ingresa un determinado tipo de paquete. Así mismo, también se informa al administrador de si el paquete ha sido descartado.

En el código 4.11 se muestra una parte del código empleado en la aplicación, para el caso concreto de los paquetes EIGRP. Para el resto de protocolos, el procedimiento es análogo al mostrado.



Código 4.11.- Parte del código de la aplicación de mitigación de ataques por envenenamiento de tablas de rutas, en la que se obtiene si el paquete procesado es EIGRP.

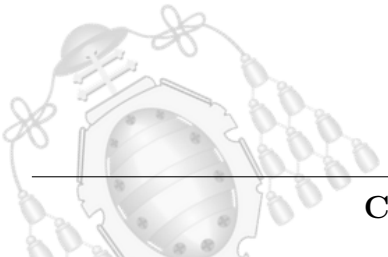
```
ip = pkt.get_protocol(ipv4.ipv4)
proto = ip.proto
if proto == 88: #Si el protocolo pertenece a EIGRP
    self.logger.debug("EIGRP PACKET RECEIVED. PORT %s",in_port)
    if in_port not in Ports_enabled: #Comprobacion si el puerto esta habilitado para recibir
        trafico EIGRP
        self.logger.debug("WARNING: PORT NOT ALLOWED TO RECEIVE EIGRP
            PACKET. DROPPING")
    return
```

4.4.6.2.- Prueba de realización del ataque con éxito

En este apartado se pretende poner de manifiesto el problema que puede suponer el descontrol de tráfico de enrutamiento en una red así como una mala configuración de estos protocolos, sin las medidas de seguridad. De todos los protocolos expuestos, se expone como ejemplo RIP, aunque para los restantes el procedimiento sería análogo.

El escenario de red en el que se lleva a cabo este ataque es el expuesto en el apartado 4.4.3. El objetivo es engañar a los routers que ejecutan el protocolo RIP, en su versión 2, sin ningún tipo de autenticación y sin medidas de restricción de tráfico en aquellos puertos de switches que no albergan routers tras ellos. El atacante se hace pasar por una puerta de enlace y que da acceso a la red 120.10.20.0/24. Esto supone suplantar al router R2.

En todos los R1 y R2 se configura RIPv2. Después se puede observar mediante la aplicación Wireshark, ejecutada desde la máquina atacante, el envío de tráfico RIPv2 por la red, fruto de la comunicación entre dispositivos, tal y como se observa en la figura 4.22.



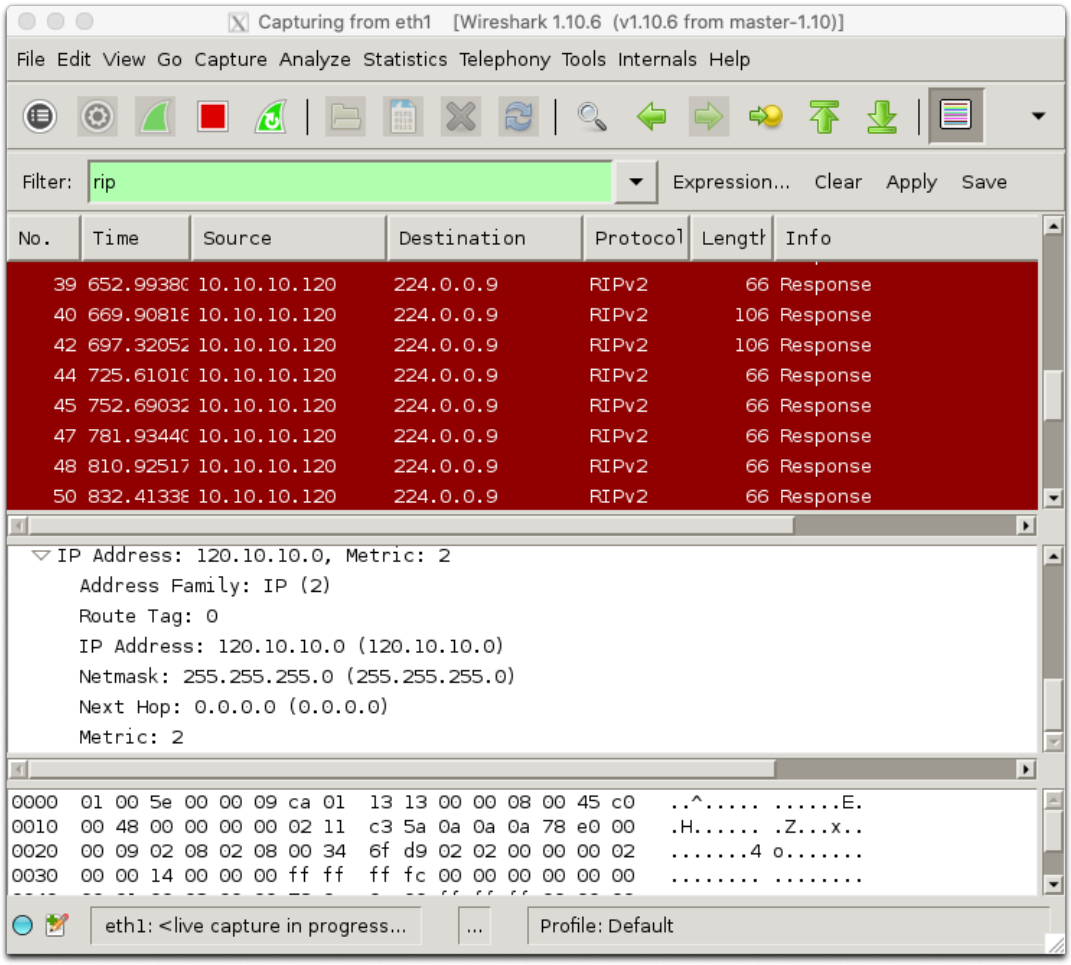


Figura 4.22.- Tráfico de paquetes RIPv2 entre las puertas de enlace legítimas.

En cada paquete RIP recibido, podemos obtener información muy valiosa, como por ejemplo las redes que anuncia cada router y su métrica. Estos datos son los que el atacante necesita para poder replicar y generar paquetes fraudulentos. En concreto, como se refleja la figura 4.23 la puerta de enlace 10.10.10.120 anuncia la red 120.10.10.0/24 con una métrica de 2. A partir de ello, se procede por parte del atacante a generar los paquetes RIP.

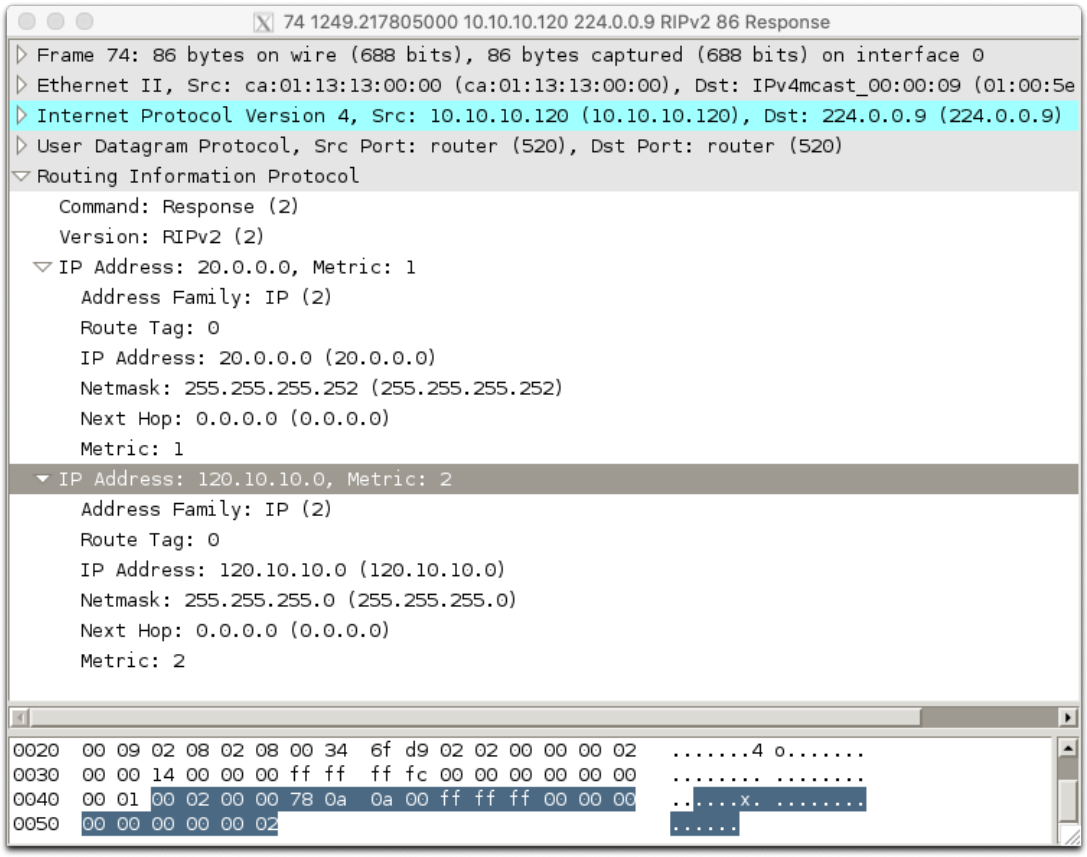


Figura 4.23.- Inspección de paquete RIPv2.

Tras el intercambio de mensajes, cada router ha aprendido las rutas anunciadas por el otro. Para comprobar esto, se ejecuta el comando **`show ip route`** en el router R1, que es la puerta de enlace de la red y se observa la siguiente información, mostrada en la figura 4.24

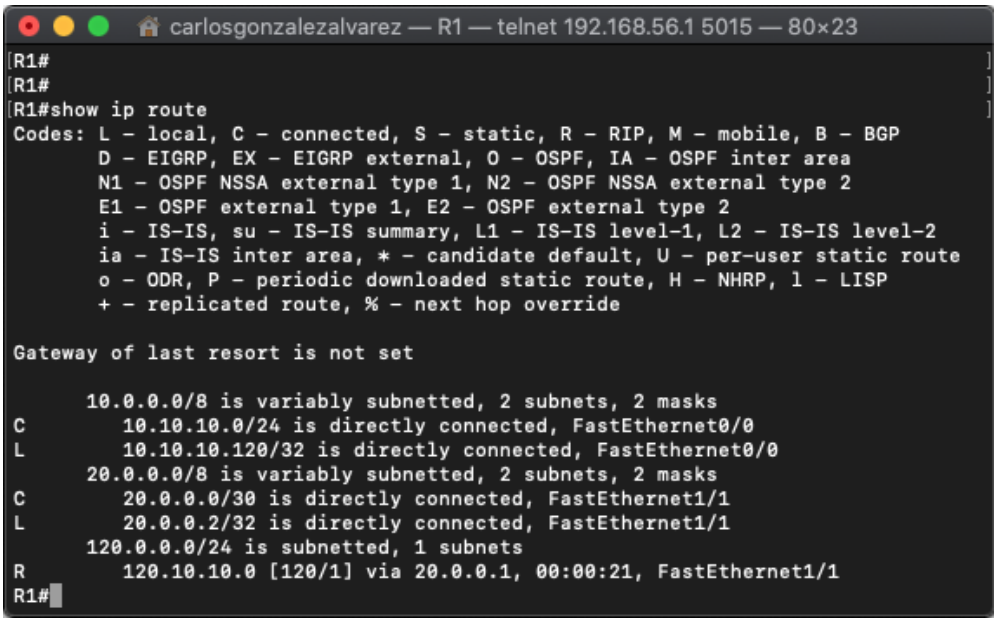


Figura 4.24.- Rutas aprendidas por el router R1, mostradas tras la ejecución del comando «show ip route».

En la última línea se puede observar como ha aprendido, por medio de RIP, que se puede acceder a la red 120.10.20.0/24 a través de la dirección 20.0.0.1.

Para la generación de los paquetes RIP fraudulentos, se hace uso de Scapy. Se crea un script de Python que se encarga de lanzar, en cada ejecución, un paquete con los siguientes parámetros:

- IP de origen: 10.10.10.5 (la propia del atacante).
- IP de destino: 224.0.0.9. La multicast de RIP.
- Puerto de origen y destino UDP: 520.
- Red a anunciar: 120.10.20.0.
- Métrica: 1. Interesa poner siempre la más baja posible, para que el la puerta de enlace considere el atacante como camino más corto y, por tanto, el prioritario.

Con todos estos parámetros, se procede a construir el script en el programa Scapy, cuyo resultado final se traduce en el código 4.12.

Código 4.12.- Script para la generación de paquetes RIPv2 fraudulentos.

```
def main():
    try:
        packet =
            scapy.Ether()/scapy.IP(src='10.10.10.5',dst="224.0.0.9")/scapy.UDP(sport=520,
            dport=520)/RIP(cmd=2, version=2)
        packet = packet/RIPEntry(AF="IP", RouteTag=0, addr="120.10.20.0",
            mask="255.255.255.0", nextHop="0.0.0.0", metric=1)
        scapy.sendp(packet, iface="eth1")
    except Exception as e:
        print(e)
        sys.exit()
```

Se ejecuta el script que genera el paquete RIPv2 fraudulento en la máquina atacante, el cual se propaga por toda la red. El paquete generado se puede observar en la figura 4.25, en una captura realizada por medio del programa Wireshark.

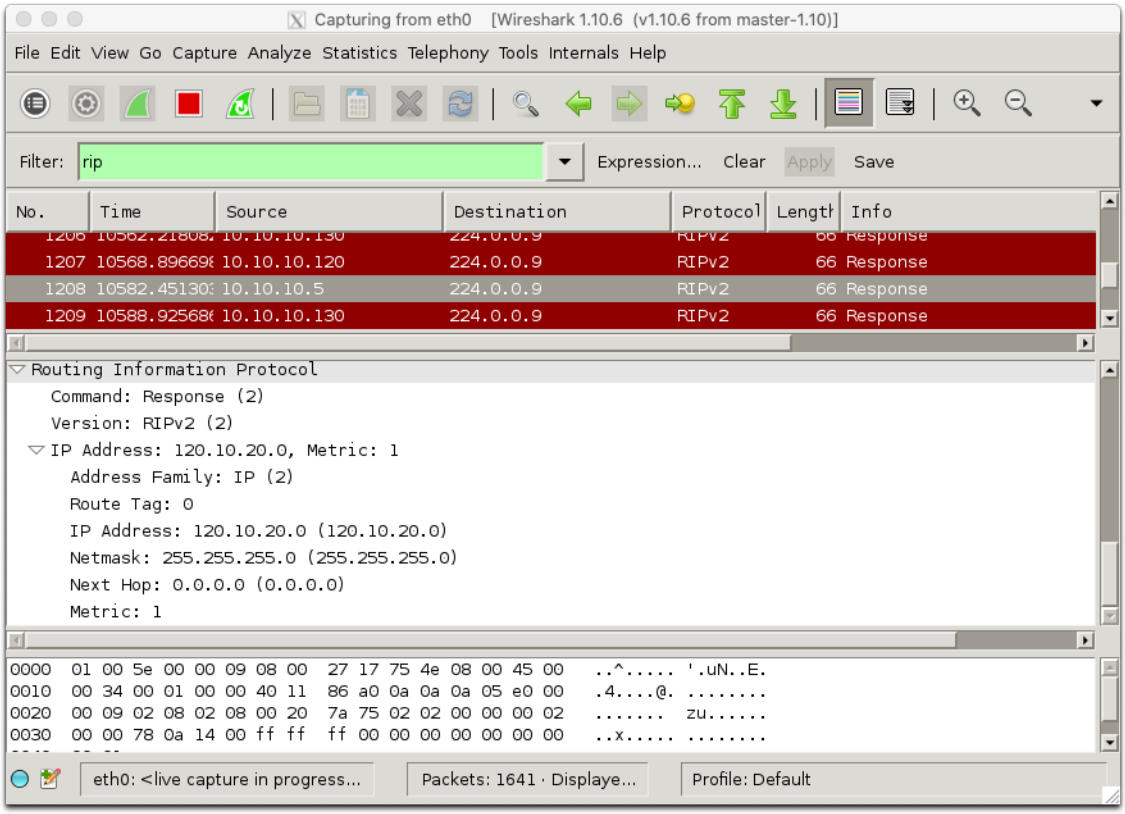


Figura 4.25.- Paquete RIPv2 fraudulento capturado por Wireshark en la máquina atacante.

A continuación, en el router R1, se comprueba de nuevo mediante el comando **`show ip route`** si ha aprendido nuevas rutas. En la imagen 4.26 se ha señalado en rojo que ha aprendido la nueva ruta, hacia la red 120.10.10.0/24 a través del atacante, la IP 10.0.0.5, con una métrica inferior a la ruta legítima. Ahora el router, cada vez que tenga que enviar algún paquete cuyo destino es la red 120.10.10.0/24 lo envía a través del atacante.

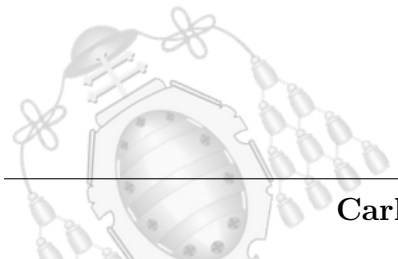
Por parte del atacante, es necesario que se envíen paquetes RIPv2 de manera constante para que el router no deje de olvidar las rutas.

El atacante envía cada 30 segundos de forma periódica mensajes RIPv2 para mostrar que se encuentra activo y el router R1 siga aprendiendo las rutas falsificadas. Esto es posible realizar de manera automática, modificando la instrucción de envío de paquetes (*sendp*) del código 4.12 por la siguiente:

```
scapy.sendp(packet, iface="eth1", loop=1, inter=30)
```

Con la instrucción «loop» se indica que la repetición del envío de manera indefinida, y con el parámetro «inter» que sea cada 30 segundos.

Como se observa en la imagen 4.26, la ruta para acceder a la red 120.10.20.0/24 es, únicamente, a través del atacante.



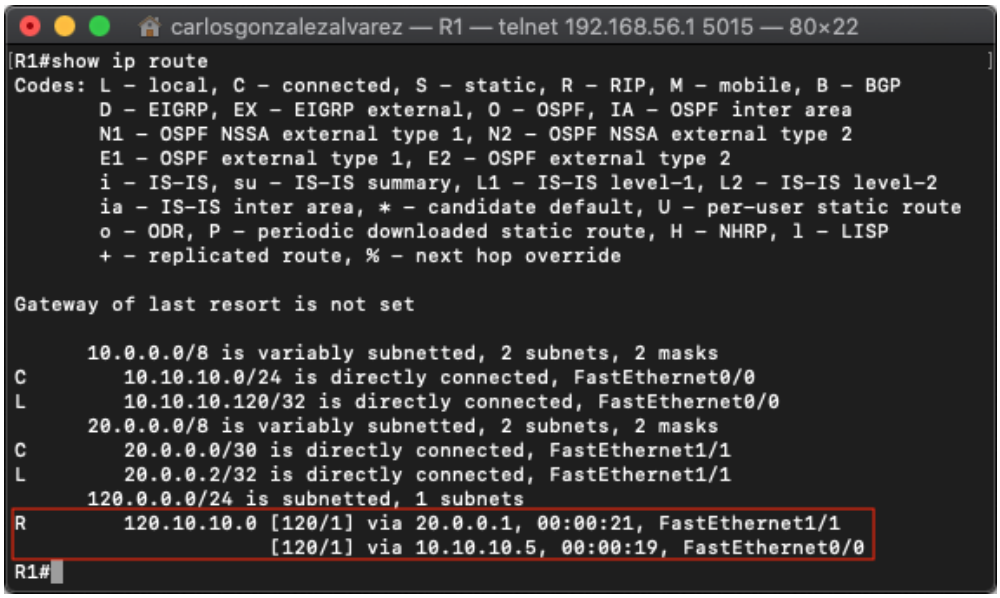


Figura 4.26.- Rutas aprendidas por el router R1 tras el ataque RIP, mostradas tras la ejecución del comando 'show ip route'.

Para comprobar que, en efecto, la ruta está envenenada y los paquetes puede ser capturados por el atacante, se procede a realizar una prueba de este hecho. Se incluye un PC de prueba en la red 120.10.10.0/24, con dirección IP 120.10.10.10. El PC objetivo del ataque, intenta enviar tráfico hacia ese ordenador y el router desvía ese tráfico hacia el atacante, puesto que en su tabla de rutas así está indicado. Para simular tráfico, basta con hacer un simple ping desde el equipo atacado hacia el PC con IP 120.10.10.10. En la máquina atacante, se ejecuta Wireshark con el fin de comprobar si los paquetes ICMP se reciben aunque no fuese su destinatario legítimo.

Se ejecutan cinco pings y el resultado en la máquina atacante es el que refleja la figura 4.27.

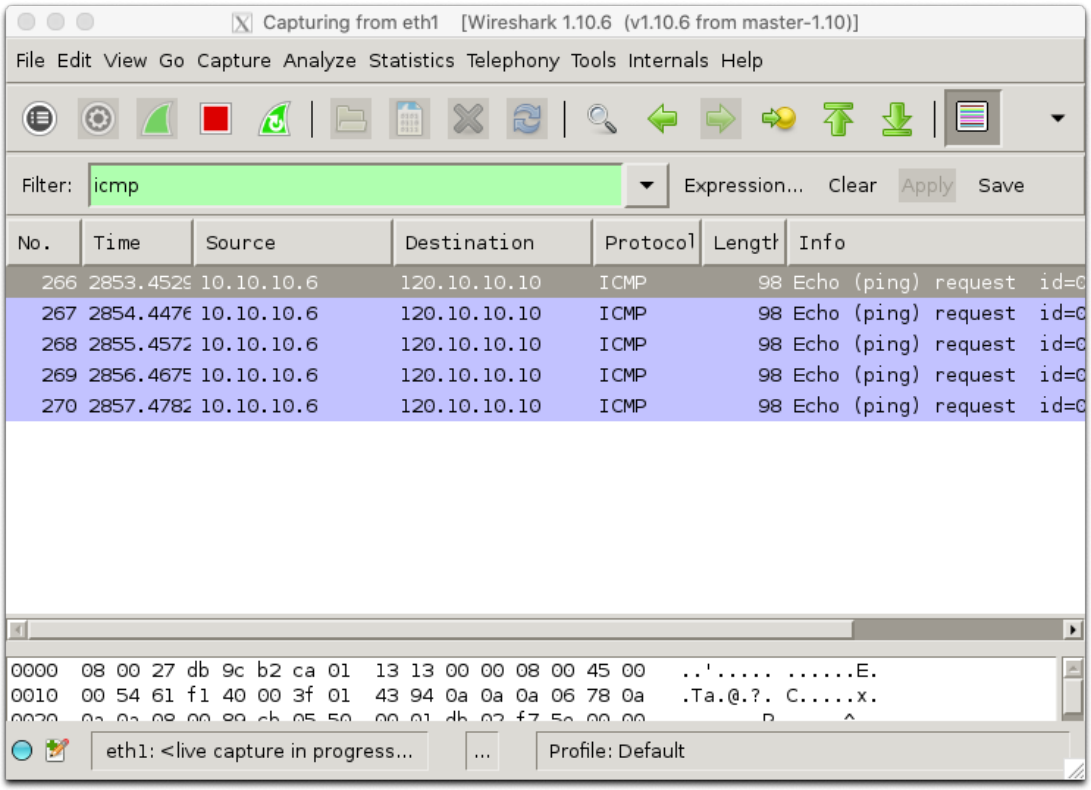


Figura 4.27.- Paquetes ICMP recibidos en la máquina atacante.

Al no haber enrutado el tráfico hacia la red legítima, los pings ejecutados por la víctima no son respondidos, mostrándose el mensaje «timeout» correspondiente en la consola de comandos:

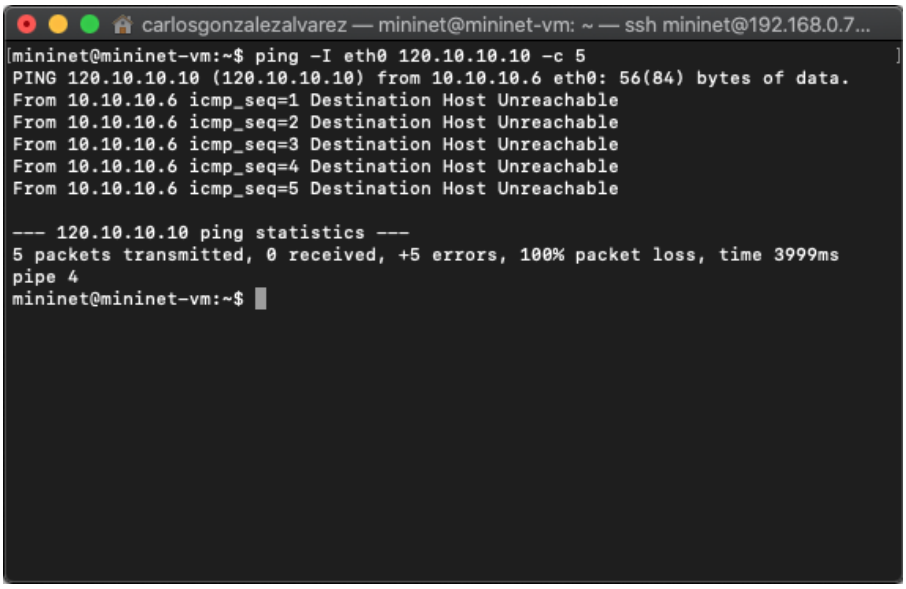


Figura 4.28.- Pings sin respuesta en la máquina víctima, consecuencia del desvío de tráfico de manera ilegítima.

Para completar el ataque de forma exitosa, el atacante podría enrutar todo el tráfico a la red legítima y actuaría como intermediario, interceptando todo el tráfico de ida y vuelta. Es lo que se conoce como ataque *Man in the Middle*

4.4.6.3.- Prueba de detención del ataque

Para la comprobación de la aplicación, se parte del escenario anteriormente mencionado, en las mismas condiciones. Una vez se comprueba que la ruta ha sido envenenada y que todos los mensajes que tienen como destino la red 120.10.20.0/24 pasan a través del atacante, se inicia la aplicación.

Se especifica en la variable **Ports_enabled** que el puerto 13 permita tráfico de protocolos de enrutamiento, denegándolo para el resto de puertos del switch.

```
Ports_enabled = [13]
```

Con ello, todos los mensajes que provienen del atacante, conectado al puerto 4, deben descartarse y no afectar a las tablas de rutas de los dispositivos de enrutamiento legítimos. Se debe mostrar por el controlador una alerta al usuario, indicando que se

ha recibido un paquete RIP por el puerto 4 y que dicho puerto no está autorizado para recibir dicho tráfico, como se observa en la figura 4.29

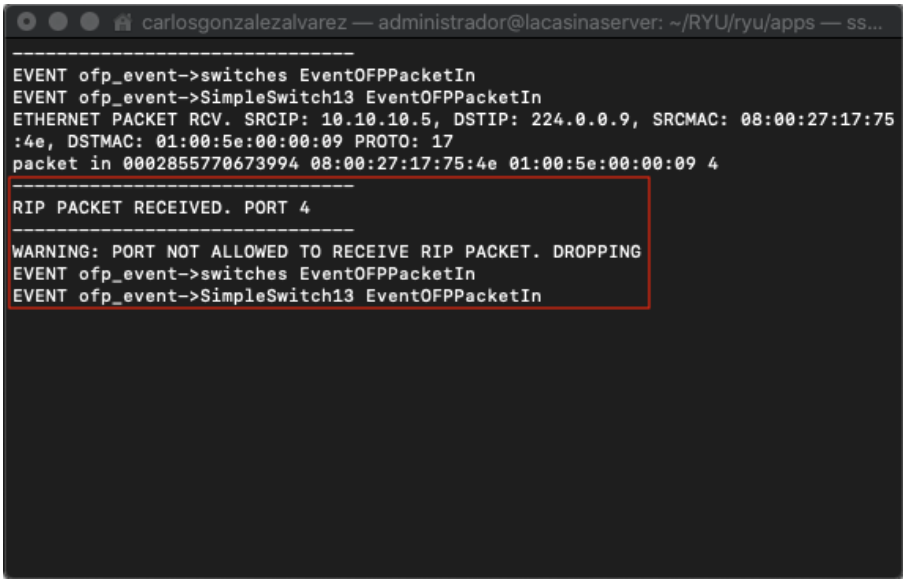


Figura 4.29.- Mensaje de advertencia en el controlador de la recepción de un paquete RIP por un puerto no autorizado.

Con esto, el router R1 deja de recibir mensajes RIP falsificados. Al cabo de tres minutos, el router al no recibir paquetes RIP del atacante considera esa ruta como no válida y la elimina de su tabla, como bien se expone en la figura 4.30, en donde ya no se refleja la red 120.10.20.0/24 asociada al usuario malicioso, el 10.10.10.5.

Figura 4.30.- Mensaje de advertencia en el controlador de la recepción de un paquete RIP por un puerto no autorizado.

Se verifica por tanto, el correcto funcionamiento de la aplicación y su gran utilidad en un escenario real con un ataque correctamente ejecutado.

4.4.7.- Mejoras

Como se ha expuesto en anteriores ataques, toda aplicación es susceptible de mejoras que afecten bien al rendimiento, a la facilidad de uso o a las opciones de configuración dependiendo de los casos de uso. A las características y posibilidades que ofrece la aplicación de inicio, se añaden una serie de mejoras tanto en el plano de la seguridad como de configuración.

4.4.7.1.- Discriminación de puertos por dispositivo

En una red definida por software, un mismo controlador puede manejar varios switches de manera simultánea. De igual forma, una red puede estar gobernada por varios controladores, bien de manera simultánea, o con un controlador principal y varios que tienen que estar disponibles de manera inmediata en caso de indisponibilidad del controlador principal.

Las directrices de puertos mencionadas para este tipo de ataques se consideran para el caso de un único switch gobernado por un único controlador. Sin embargo, en caso de que en la red existiesen más de un switch, las políticas se aplicarían de manera idéntica para cada uno de ellos. Esta rigidez no permite que para cada switch se apliquen políticas distintas, máxime dada la heterogeneidad de este tipo de redes en cuanto a equipamiento y disposición de los mismos. Ello dificulta enormemente una de las cualidades más importantes de la filosofía SDN, que es la escalabilidad de la red y su versatilidad.

Para esta aplicación en concreto, se supone un escenario conformado por dos switches, A y B, gobernados por un mismo controlador, C:

- El switch A tiene conectados en su puerto 1 un router.
- El switch B tiene conectado en el puerto 2 otro router.

Figura 4.31.- Hipotético escenario con dos puertas de enlace distintas en cada switch.

Por tanto, resulta lógico que para el switch A se permita tráfico de protocolos de enrutamiento en el puerto 1 y, en el caso del switch B para el puerto 2 únicamente. Tal y como está construida la aplicación, esto solamente sería posible declarando los puertos 1 y 2 como accesibles a este tipo de tráfico. El resultado sería que el switch A tendría un puerto, el número 2, libre de restricciones y el B el puerto 1, de manera innecesaria. Una configuración mal hecha y que compromete a la red. El problema se agravaría si se contase con un mayor número de switches en la red.

Es por ello que debe distinguirse entre switches. Para hacer esto posible, se hace uso de un identificador único con el que cuenta cada switch de la red, denominado **dpid** (datapath ID). En cada paquete que es recibido por el controlador, entre otras informaciones, es posible extraer este identificador, que indica el switch que ha enviado dicho paquete. Con ello, resulta relativamente sencillo implementar esta mejora.

Basta con conocer los identificadores de cada switch y comprobar qué puertos tiene permitidos. Esto implica modificar, por un lado, la variable de puertos definida con anterioridad, que consistía en una lista, por un diccionario. Las claves únicas van a ser los identificadores del switch. Para el ejemplo de escenario mencionado con anterioridad, la configuración de la variable **Ports_enabled** sería como sigue:

```
#Se distinguen los switches en base a su identificador unico
Ports_enabled = {
                    "datapath_A": [1],
                    "datapath_B": [2]
                }
```

Por otro lado en el código 4.11, durante la comprobación, debe consultarse la lista de puertos asociados al switch del que provenga el paquete. El cambio se muestra en el código 4.13, concretamente en la línea 6.

Código 4.13.- Parte del código de la aplicación de mitigación de ataques por envenenamiento de tablas de rutas, con las mejoras que distinguen entre switches.

```
    ip = pkt.get_protocol(ipv4.ipv4)
    proto = ip.proto
    dpid = pkt.get(dpid)
        if proto == 88: #Si el protocolo pertenece a EIGRP
            self.logger.debug("EIGRP PACKET RECEIVED. PORT %s",in_port)
            if in_port not in Ports_enabled[dpid]: #Comprobacion si el puerto esta habilitado para
                recibir trafico EIGRP
                self.logger.debug("WARNING: PORT NOT ALLOWED TO RECEIVE EIGRP
                    PACKET. DROPPING")
                return
```

4.4.7.2.- Detección de ataques de denegación de servicio

Como se ha comentado en el apartado 4.4.6.1, todos los paquetes recibidos que pertenezcan a los protocolos de enrutamiento anteriormente mencionados, son empaquetados y enviados al controlador para su inspección. Si un atacante tiene conocimiento de este proceder, puede aprovechar este funcionamiento para generar y enviar grandes volúmenes de datos, convirtiéndolo en un ataque por denegación de servicio hacia el controlador.

Dejar fuera de servicio al controlador en una SDN, a menos que existan controladores de respaldo, puede provocar bastantes problemas en el rendimiento y correcto funcionamiento de la red. Dependiendo de la configuración que se haya establecido de partida, si un switch deja de tener comunicación con el controlador, puede actuar de dos maneras:

1. El switch toma el control y pase a funcionar como un conmutador en una red tradicional.
2. El switch entra en modo «emergencia». Los flujos presentes en un switch pueden llevar asociado un campo denominado «bit de emergencia». De encontrarse activo, éste no se eliminaría en el supuesto caso de que el switch perdiese la conexión con el controlador. El resto de flujos que no tuviesen ese bit activo, se eliminarían.

Con esta mejora lo que se persigue es evitar este tipo de situaciones que se aprovechan del funcionamiento de la aplicación. La técnica que se utiliza para ello es la misma que la implementada en el ataque de escaneo: detectar si se recibe un volumen importante de tráfico de un mismo tipo, por un puerto concreto. Las acciones llevadas a cabo pasan por establecer una regla en el switch que automáticamente descarte ese tipo de tráfico, evitando así su llegada al controlador.

El proceso de detección de un posible ataque es análogo al llevado a cabo para la detección de escaneos. Se lleva un registro del número de paquetes recibidos por

unidad de tiempo. De exceder un determinado umbral, se llevan a cabo las acciones correspondientes.

Para instaurar la regla que descarte los paquetes en el switch, se crea una función, **drop_routing_traffic**, análoga a la expuesta en el código 4.3 pero con una diferencia, que en los argumentos que se introducen en dicha función se indica también el id del protocolo correspondiente.

En el código 4.14 se muestra la implementación para detectar ataques por denegación de servicio utilizando paquetes EIGRP. Para este caso, si el tiempo transcurrido entre la llegada de paquetes de estos protocolos por un mismo puerto es inferior a 200 milisegundos, se aumenta en una unidad el contador asociado. De rebasar éste 100, se activa el flujo para descartar dicho tráfico entrante por ese puerto en el switch. Adicionalmente, se muestran por pantalla en el controlador mensajes indicativos de alerta.

Código 4.14.- Código para detectar la llegada de paquetes EIGRP y si constituyen un ataque por denegación de servicio

```
if proto == 88:
    self.logger.debug("EIGRP PACKET RECEIVED. PORT %s",in_port)
        if in_port not in Ports_enabled[dpid]:
            if in_port not in eigrp_counter:
                eigrp_counter[in_port]={'counter':1,'timestamp':current_milli_time()}
            else:
                deltha = int(current_milli_time()) - int(proto_counter[in_port]['timestamp'])
                proto_counter[in_port]['timestamp'] = current_milli_time()
            if deltha < 200:
                proto_counter[in_port]['counter'] += 1
            if (proto_counter[in_port]['counter']) >100:
                self.logger.debug("WARNING: EIGRP FLOOD ATTACK DETECTED")
                self.drop_routing_traffic(datapath,0x0058,in_port)
                return
        self.logger.debug("WARNING: PORT NOT ALLOWED TO RECEIVE EIGRP
            PACKET. DROPPING")
        return
```

4.4.7.3.- Fichero de configuración

Siguiendo la línea de aplicaciones anteriores y con el fin de evitar la inclusión de variables en el propio código, se opta por derivar todas ellas a un fichero de configuración externo, teniendo en cuenta todas las directrices y mejoras que han ido progresivamente implantándose. El formato de dicho fichero es, al igual que las anteriores, en formato YAML.

Todas estas opciones de configuración están contenidas bajo la etiqueta de **PROTOCOLS_MODULE**. Con el fin de distinguir entre switches, como clave primaria está el identificador único de cada elemento de red.

Las variables de configuración que el usuario puede modificar son:

- **Ports_enabled**: Vector que contiene los números de puertos por los que está permitido el paso del tráfico de enrutamiento.
- **deltha_time**: Tiempo máximo, en milisegundos, que puede transcurrir entre llegada de paquetes del mismo tipo de tráfico de enrutamiento, para que no sea considerado como una tormenta o ataque por denegación de servicio.
- **max_count**: Cuenta máxima a partir de la cual se aplican las medidas de descarte de tráfico en el switch.

A continuación se muestra un ejemplo de fichero de configuración para una red con dos switches:

```
0002855770673994: #Switch-1
  PROTOCOLS_MODULE:
    Ports_enabled: [15]
    deltha_time: 50 #In miliseconds
    max_count: 10 #Number of matches

0143080643644224: #Switch-2
  PROTOCOLS_MODULE:
    Ports_enabled: [4, 13]
    deltha_time: 500 #In miliseconds
    max_count: 20 #Number of matches
```

5. Conclusiones y futuras líneas de investigación

A lo largo de este proyecto, se ha trabajado en el ámbito de las redes definidas por software (SDN), principalmente en una de sus tres partes, que es la capa de aplicación. Se han desarrollado diversas aplicaciones con el objetivo mejorar la seguridad perimetral en el ámbito de este nuevo concepto de red, surgido en el año 2009 y los resultados, a la vista de lo expuesto en este documento, han sido satisfactorios.

La versatilidad y la programabilidad de las SDN permiten desarrollar aplicaciones específicas para casi cualquier ámbito de la red, y este ha sido el principal catalizador de este proyecto. Sin embargo, a pesar de los buenos resultados obtenidos, aún son muchos los puntos a desarrollar e investigar acerca de este tema concreto.

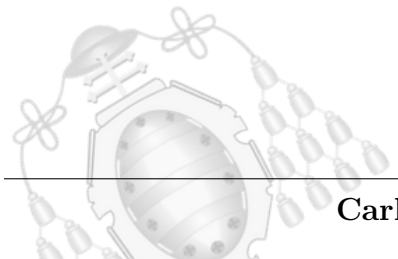
El principal objetivo a corto plazo pasa por desplegar estas aplicaciones en un entorno real de trabajo, con equipamiento físico y no virtualizado. A pesar de que todo este presente trabajo ha sido ejecutado íntegramente en un entorno virtual, bastante próximo a uno real, los resultados que se obtienen, sobre todo en términos de rendimiento, no pueden ser del todo extrapolados a un entorno físico real. Este despliegue en entornos con equipamiento real, también obedece a otra de las líneas de trabajo futuro, que es la utilización de equipamiento de otros fabricantes y que permitan su integración con OpenFlow, como Cisco, HP o Huawei entre otros grandes fabricantes.

Otro de los puntos a investigar es la mejora de la eficiencia de estas aplicaciones que, en su mayoría, delegaban todo el trabajo en el controlador, sobrecargándolo de tareas y ralentizando el procesamiento de paquetes en la red. Ello pasa por delegar todas, o gran parte de estas tareas en los switches de la red, a base de creación y eliminación de reglas en sus correspondientes tablas de flujo. También aprovechar de alguna manera las métricas y contadores que proporcionan las tablas de flujo de los switches.

Existen también otros puntos muy interesantes de cara a futuras investigaciones en este ámbito, y que también atañan a otros aspectos de las redes SDN:

- Despliegue en redes con más de un controlador, dividiendo la carga de trabajo de las aplicaciones entre varios controladores de la red y reduciendo el uso de recursos, así como dotar de mayor robustez a la red en cuestiones de disponibilidad del controlador.
- En cuanto a Open vSwitch, utilizado en este proyecto, explotar las posibilidades que ofrece OVSDB para la obtención de la configuración en los equipos, como por ejemplo las VLANs asociadas a sus puertos y también la capacidad de modificar, directamente desde las aplicaciones, estos parámetros.
- Explorar las posibilidades de utilización de estas aplicaciones para la seguridad en protocolos VoIP.
- Desarrollo de interfaces gráficas para las aplicaciones, como sucede en otros controladores como puede ser el caso de OpenDayLight.
- Exploración de otros controladores que no sean Ryu, tales como ONOS, OpenDayLight o POX.

Aunque son muchas las tareas a explotar en este ámbito, los resultados obtenidos han superado las expectativas iniciales de este proyecto. Las SDN se han consolidado en los últimos años, aunque no han estado exentas de los problemas de seguridad que adolecen las redes tradicionales. Sin embargo, las posibilidades que ofrecen este tipo de aplicaciones, sobre todo en una cuestión tan capital como es la seguridad y que día a día es más creciente, convierten a esta tecnología de red en un claro candidato a sustituir las redes actuales.



Anexo I. Aplicación de switch de capa 2.

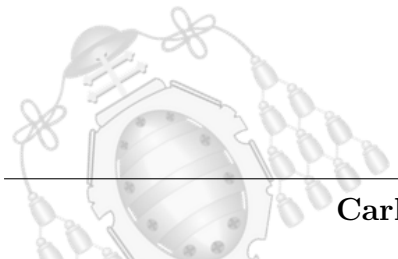
Código 5.1.- Aplicación por defecto de un switch de capa 2.

```
import time
import os
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types
from ryu.controller import dpset

class SimpleSwitch13(app_manager.RyuApp):
OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
_CONTEXTS = {'dpset': dpset.DPSet}
def __init__(self, *args, **kwargs):
super(SimpleSwitch13, self).__init__(*args, **kwargs)
self.mac_to_port = {}
self.dpset = kwargs['dpset']

def send_port_desc_stats_request(self, datapath):
ofp_parser = datapath.ofproto_parser
req = ofp_parser.OFPPortDescStatsRequest(datapath, 0)
datapath.send_msg(req)

@set_ev_cls(ofp_event.EventOFPPortDescStatsReply, MAIN_DISPATCHER)
def port_desc_stats_reply_handler(self, ev):
ports = []
print(ev.msg.body)
for p in ev.msg.body:
ports.append('port_no=%d hw_addr=%s name=%s config=0x%08x '
'state=0x%08x curr=0x%08x advertised=0x%08x '
'supported=0x%08x peer=0x%08x curr_speed=%d '
'max_speed=%d' %
(p.port_no, p.hw_addr,
```

```
p.name, p.config,
p.state, p.curr, p.advertised,
p.supported, p.peer, p.curr_speed,
p.max_speed))
self.logger.debug('OFPPortDescStatsReply received: %s', ports)

@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
datapath = ev.msg.datapath
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
match = parser.OFPMatch()
actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
ofproto.OFPCML_NO_BUFFER)]
self.add_flow(datapath, 0, match, actions)

def add_flow(self, datapath, priority, match, actions, buffer_id=None):
ofproto = datapath.ofproto
parser = datapath.ofproto_parser

inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
actions)]
if buffer_id:
mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
priority=priority, match=match,
instructions=inst)
else:
mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
match=match, instructions=inst)
datapath.send_msg(mod)

def add_custom_flow(self,datapath,port,dest,vlanid):
ofproto = datapath.ofproto
parser = datapath.ofproto_parser

#match = parser.OFPMatch(in_port=port,eth_type=0x0800,vlan_vid=vlanid)
match = parser.OFPMatch(vlan_vid=vlanid)
actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER)]
```

```
self.add_flow(datapath, 2, match, actions)

def send_port_mod(self, datapath, port_no, hard_addr, mask, advertise):
ofp = datapath.ofproto
ofp_parser = datapath.ofproto_parser
config = 1
req = ofp_parser.OFPPortMod(datapath, port_no, hard_addr, config,
    ofp.OFPPC_PORT_DOWN, advertise)
datapath.send_msg(req)

@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
#Aqui es donde debemos actuar con la vaina
def _packet_in_handler(self, ev):
# If you hit this you might want to increase
# the "miss_send_length" of your switch
if ev.msg.msg_len < ev.msg.total_len:
self.logger.debug("packet truncated: only %s of %s bytes",
ev.msg.msg_len, ev.msg.total_len)
msg = ev.msg
datapath = msg.datapath
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
in_port = msg.match['in_port']
port_info = self.dpset.get_port(datapath.id, in_port)
hw_addr = port_info[1]
pkt = packet.Packet(msg.data)
eth = pkt.get_protocols(ethernet.ethernet)[0]

if eth.ethertype == ether_types.ETH_TYPE_LLDP:
# ignore lldp packet
return
dst = eth.dst
src = eth.src
dpid = format(datapath.id, "d").zfill(16)
self.mac_to_port.setdefault(dpid, {})
print(pkt)
self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)
```

```
# learn a mac address to avoid FLOOD next time.
self.mac_to_port[dpid][src] = in_port

if dst in self.mac_to_port[dpid]:
out_port = self.mac_to_port[dpid][dst]
else:
out_port = ofproto.OFPP_FLOOD

actions = [parser.OFPActionOutput(out_port)]

# install a flow to avoid packet_in next time
if out_port != ofproto.OFPP_FLOOD:
match = parser.OFPMatch(in_port=in_port, eth_dst=dst, eth_src=src)
# verify if we have a valid buffer_id, if yes avoid to send both
# flow_mod & packet_out
if msg.buffer_id != ofproto.OFP_NO_BUFFER:
self.add_flow(datapath, 1, match, actions, msg.buffer_id)
return
else:
self.add_flow(datapath, 1, match, actions)
data = None
if msg.buffer_id == ofproto.OFP_NO_BUFFER:
data = msg.data

out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
in_port=in_port, actions=actions, data=data)
datapath.send_msg(out)
```



Anexo II. Aplicación para detección de escaneos en red

Código 5.2.- Aplicación para detección de escaneos en red

```
from ___future___ import unicode_literals
import time
import os
import ipaddress as ipad
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import ipv4
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types
from ryu.controller import dpset
from ryu.lib.packet import udp
import yaml


ARP_COUNTER = {}
vlans_dict = {}
vlans_ports = {}
device_ip = '15.0.0.2'
gateway_ip = ''
gateway_mac = ''
gateway_port = 0
network_ip = ''
subnet_mask = ''
config_file = {}
ospf_counter = {}
eigrp_counter = {}


class SimpleSwitch13(app_manager.RyuApp):
OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
_CONTEXTS = {'dpset': dpset.DPSet}
def __init__(self, *args, **kwargs):
```

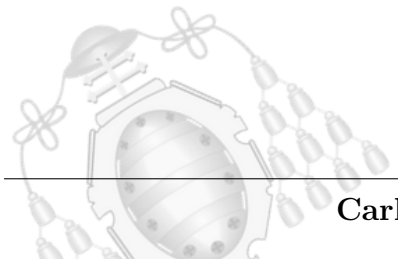
```
super(SimpleSwitch13, self).__init__(*args, **kwargs)
self.mac_to_port = {}
self.dpset = kwargs['dpset']

def update_yaml_file(self,dpid):
global config_file, network_ip, subnet_mask, gateway_ip, device_ip
a_yaml_file = open("config.yaml")
config_file = yaml.load(a_yaml_file, Loader=yaml.FullLoader)
network_ip = config_file[dpid]['PVLAN_MODULE']['Network_IP']
subnet_mask = config_file[dpid]['PVLAN_MODULE']['Subnet_Mask']
gateway_mac = config_file[dpid]['PVLAN_MODULE']['Gateway_MAC']
gateway_ip = config_file[dpid]['PVLAN_MODULE']['Gateway_IP']

def send_port_desc_stats_request(self, datapath):
ofp_parser = datapath.ofproto_parser

req = ofp_parser.OFPPortDescStatsRequest(datapath, 0)
datapath.send_msg(req)

#Evento al recibir mensaje estado puertos
@set_ev_cls(ofp_event.EventOFPPortDescStatsReply, MAIN_DISPATCHER)
def port_desc_stats_reply_handler(self, ev):
global vlans_dict
global vlans_ports
self.get_vlan_ports(device_ip)
ports = []
for p in ev.msg.body:
if p.name in vlans_dict:
vlans_ports[p.port_no]= {'port_name':p.name,'vlan_tag':vlans_dict[p.name]['vlan_id']}

@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
datapath = ev.msg.datapath
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
match = parser.OFPMatch()
actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
ofproto.OFPCML_NO_BUFFER)]
```

```
self.add_flow(datapath, 0, match, actions)
self.logger.debug('Features request successfully sent')
self.send_port_desc_stats_request(datapath)
#self.add_custom_flow(datapath,'ca:01:13:13:00:00')

def add_flow(self, datapath, priority, match, actions, buffer_id=None):
ofproto = datapath.ofproto
parser = datapath.ofproto_parser

inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
actions)]
if buffer_id:
mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
priority=priority, match=match,
instructions=inst)
else:
mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
match=match, instructions=inst)
datapath.send_msg(mod)

def add_custom_flow(self,datapath,dest):
ofproto = datapath.ofproto
parser = datapath.ofproto_parser

match = parser.OFPMatch(eth_dst=dest)
actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER)]
self.add_flow(datapath, 2, match, actions)

def drop_arp_traffic(self,datapath,port):
parser = datapath.ofproto_parser
ofproto = datapath.ofproto
match = parser.OFPMatch(eth_type=ether_types.ETH_TYPE_IP,ip_proto=proto,
    in_port=port)
actions = [parser.OFPInstructionActions(ofproto.OFPIT_CLEAR_ACTIONS,[])]
mod = parser.OFPFlowMod(datapath=datapath,
out_port=ofproto.OFPP_ANY,
out_group=ofproto.OFPG_ANY,
match=match, instructions=actions)
```

```
#self.add_flow(datapath, 2, match, actions)
datapath.send_msg(mod)

def send_port_mod(self, datapath, port_no, hard_addr, mask, advertise):
ofp = datapath.ofproto
ofp_parser = datapath.ofproto_parser
config = 1
req = ofp_parser.OFPPortMod(datapath, port_no, hard_addr, config,
    ofp.OFPPC_PORT_DOWN, advertise)
datapath.send_msg(req)

def shutdown_port(self, datapath, port_no, hard_addr):
ofp = datapath.ofproto
ofp_parser = datapath.ofproto_parser
config = 1
req = ofp_parser.OFPPortMod(datapath, port_no, hard_addr, config,
    ofp.OFPPC_PORT_DOWN, datapath.ofproto.OFPPF_PAUSE)
datapath.send_msg(req)

@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
# If you hit this you might want to increase
# the "miss_send_length" of your switch
global gateway_mac
global gateway_port
global gateway_port_numbers
global ospf_counter, eigrp_counter
if ev.msg.msg_len < ev.msg.total_len:
self.logger.debug("packet truncated: only %s of %s bytes",
ev.msg.msg_len, ev.msg.total_len)
msg = ev.msg
datapath = msg.datapath
dpid = format(datapath.id, "d").zfill(16)
self.update_yaml_file(dpid)
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
in_port = msg.match['in_port']
port_info = self.dpset.get_port(datapath.id, in_port)
```

```
hw_addr = port_info[1]
pkt = packet.Packet(msg.data)
eth = pkt.get_protocols(ethernet.ethernet)[0]
dst = eth.dst
src = eth.src

if eth.ethertype == ether_types.ETH_TYPE_LLDP:
# ignore lldp packet
return
if eth.ethertype == ether_types.ETH_TYPE_IP:
ip = pkt.get_protocol(ipv4.ipv4)
srcip = ip.src
dstip = ip.dst
proto = ip.proto

global ARP_COUNTER
current_milli_time = lambda: int(round(time.time() * 1000))

#Detectar ARP
if dst == 'ff:ff:ff:ff:ff:ff':
if dpid in ARP_COUNTER:
deltha = current_milli_time() - int(ARP_COUNTER[dpid][src]['timestamp'])
ARP_COUNTER[src]['timestamp'] = current_milli_time()
if deltha < 50: #Hay escaneo
ARP_COUNTER[src]['counter'] += 1
if ARP_COUNTER[src]['counter'] >= 10:
self.logger.debug("ALERTA: Escaneo de servicios detectado en el puerto %s.",in_port)
self.send_port_mod(datapath, in_port, hw_addr, datapath.ofproto.OFPPC_PORT_DOWN,
    datapath.ofproto.OFPPF_PAUSE)
self.logger.debug("INFO: El puerto %s ha sido apagado",in_port)
if dpid not in ARP_COUNTER:
ARP_COUNTER[src]= {'counter': 1, 'timestamp':current_milli_time()}
self.logger.info('ARP-COUNTER-VALUE %s', str(ARP_COUNTER[src]['counter']))

self.mac_to_port.setdefault(dpid, {})
self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)
self.mac_to_port[dpid][src] = in_port
if dst in self.mac_to_port[dpid]:
```

```
out_port = self.mac_to_port[dpid][dst]
else:
out_port = ofproto.OFPP_FLOOD

actions = [parser.OFPActionOutput(out_port)]
# install a flow to avoid packet_in next time
if out_port != ofproto.OFPP_FLOOD:
match = parser.OFPMatch(in_port=in_port, eth_dst=dst, eth_src=src)
if msg.buffer_id != ofproto.OFP_NO_BUFFER:
self.add_flow(datapath, 1, match, actions, msg.buffer_id)
return
else:
self.add_flow(datapath, 1, match, actions)
data = None
if msg.buffer_id == ofproto.OFP_NO_BUFFER:
data = msg.data
out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
in_port=in_port, actions=actions, data=data)
datapath.send_msg(out)
```



Anexo III. Aplicación para detener ataques PVLAN

Código 5.3.- Aplicación para detener ataques PVLAN

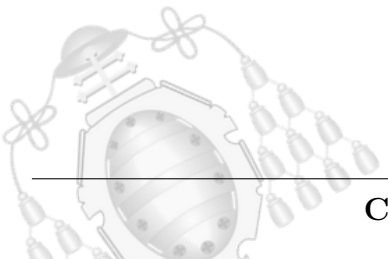
```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals
import time
import os
import ipaddress as ipad
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import ipv4
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types
from ryu.controller import dpset
from ryu.lib.packet import udp
import yaml
ARP_COUNTER = {}
vlans_dict = {}
vlans_ports = {}
device_ip = '15.0.0.2'
gateway_port = 0
config_file = {}
class SimpleSwitch13(app_manager.RyuApp):
OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
_CONTEXTS = {'dpset': dpset.DPSet}
def __init__(self, *args, **kwargs):
super(SimpleSwitch13, self).__init__(*args, **kwargs)
self.mac_to_port = {}
self.dpset = kwargs['dpset']
def update_yaml_file(self,dpid):
global config_file , network_ip, subnet_mask, gateway_ip, device_ip
a_yaml_file = open("config.yaml")
config_file = yaml.load(a_yaml_file, Loader=yaml.FullLoader)
```

```
network_ip = config_file[dpid]['PVLAN_MODULE']['Network_IP']
subnet_mask = config_file[dpid]['PVLAN_MODULE']['Subnet_Mask']
gateway_mac = config_file[dpid]['PVLAN_MODULE']['Gateway_MAC']
gateway_ip = config_file[dpid]['PVLAN_MODULE']['Gateway_IP']
def send_port_desc_stats_request(self, datapath):
ofp_parser = datapath.ofproto_parser
req = ofp_parser.OFPPortDescStatsRequest(datapath, 0)
datapath.send_msg(req)
@set_ev_cls(ofp_event.EventOFPPortDescStatsReply, MAIN_DISPATCHER)
def port_desc_stats_reply_handler(self, ev):
global vlans_dict
global vlans_ports
self.get_vlan_ports(device_ip)
ports = []
for p in ev.msg.body:
if p.name in vlans_dict:
vlans_ports[p.port_no]= {'port_name':p.name,'vlan_tag':vlans_dict[p.name]['vlan_id']}
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
datapath = ev.msg.datapath
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
match = parser.OFPMatch()
actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
ofproto.OFPCML_NO_BUFFER)]
self.add_flow(datapath, 0, match, actions)
self.logger.debug('Features request successfully sent')
self.send_port_desc_stats_request(datapath)

def add_flow(self, datapath, priority, match, actions, buffer_id=None):
ofproto = datapath.ofproto
parser = datapath.ofproto_parser

inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
actions)]
if buffer_id:
mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
priority=priority, match=match,
```

```
instructions=inst)
else:
mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
match=match, instructions=inst)
datapath.send_msg(mod)
def add_flow_pvlan(self,datapath,address,subnet_mask):
parser = datapath.ofproto_parser
ofproto = datapath.ofproto
match = parser.OFPMatch(eth_type=ether_types.ETH_TYPE_IP,
ipv4_dst=(address, subnet_mask), ipv4_src=(address, subnet_mask))
actions = [parser.OFPInstructionActions(ofproto.OFPIT_CLEAR_ACTIONS,[])]
mod = parser.OFPFlowMod(datapath=datapath,
out_port=ofproto.OFPP_ANY,
out_group=ofproto.OFPG_ANY,
match=match, instructions=actions)
datapath.send_msg(mod)
def send_port_mod(self, datapath, port_no, hard_addr, mask, advertise):
ofp = datapath.ofproto
ofp_parser = datapath.ofproto_parser
config = 1
req = ofp_parser.OFPPortMod(datapath, port_no, hard_addr, config,
    ofp.OFPPC_PORT_DOWN, advertise)
datapath.send_msg(req)
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
global gateway_mac
global gateway_port
global gateway_port_numbers
global ospf_counter, eigrp_counter
if ev.msg.msg_len < ev.msg.total_len:
self.logger.debug("packet truncated: only %s of %s bytes",
ev.msg.msg_len, ev.msg.total_len)
msg = ev.msg
datapath = msg.datapath
dpid = format(datapath.id, "d").zfill(16)
self.update_yaml_file(dpid)
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
```

```
in_port = msg.match['in_port']
port_info = self.dpset.get_port(datapath.id, in_port)
hw_addr = port_info[1]
pkt = packet.Packet(msg.data)
eth = pkt.get_protocols(ethernet.ethernet)[0]
dst = eth.dst
src = eth.src
if eth.ethertype == 335:
self.logger.debug('Paquete LLC. Mac: %s Puerto: %s ',eth.src, in_port)
gateway_mac = eth.src
gateway_port = in_port

if eth.ethertype == ether_types.ETH_TYPE_LLDP:
# ignore lldp packet
return
if eth.ethertype == ether_types.ETH_TYPE_IP:
ip = pkt.get_protocol(ipv4.ipv4)
srcip = ip.src
dstip = ip.dst
proto = ip.proto

self.mac_to_port.setdefault(dpid, {})
self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)

if eth.ethertype == ether_types.ETH_TYPE_IP:
ip = pkt.get_protocol(ipv4.ipv4)
if (dst == gateway_mac) and (dstip != gateway_ip):
self.logger.debug('WARNING: Paquete con MAC destino Puerta de enlace fraudulento')
self.add_flow_pvlan(datapath,gateway_ip,subnet_mask)
self.logger.debug('INFO: Nueva regla anadida en el switch satisfactoriamente.')

self.mac_to_port[dpid][src] = in_port
if dst in self.mac_to_port[dpid]:
out_port = self.mac_to_port[dpid][dst]
else:
out_port = ofproto.OFPP_FLOOD

actions = [parser.OFPActionOutput(out_port)]
```

```
if out_port != ofproto.OFPP_FLOOD:
match = parser.OFPMatch(in_port=in_port, eth_dst=dst, eth_src=src)
if msg.buffer_id != ofproto.OFP_NO_BUFFER:
self.add_flow(datapath, 1, match, actions, msg.buffer_id)
return
else:
self.add_flow(datapath, 1, match, actions)
data = None
if msg.buffer_id == ofproto.OFP_NO_BUFFER:
data = msg.data

out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
in_port=in_port, actions=actions, data=data)
datapath.send_msg(out)
```



Anexo IV. Aplicación para detener ataques de envenenamiento de rutas

Código 5.4.- Aplicación para detener ataques de envenenamiento de rutas

```
from __future__ import unicode_literals
import time
import os
import ipaddress as ipad
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import ipv4
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types
from ryu.controller import dpset
from ryu.lib.packet import udp
import yaml


ARP_COUNTER = {}
vlans_dict = {}
vlans_ports = {}
device_ip = '15.0.0.2'
config_file = {}
ospf_counter = {}
eigrp_counter = {}


class SimpleSwitch13(app_manager.RyuApp):
OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
_CONTEXTS = {'dpset': dpset.DPSet}
def __init__(self, *args, **kwargs):
super(SimpleSwitch13, self).__init__(*args, **kwargs)
self.mac_to_port = {}
self.dpset = kwargs['dpset']


def update_yaml_file(self,dpid):
```

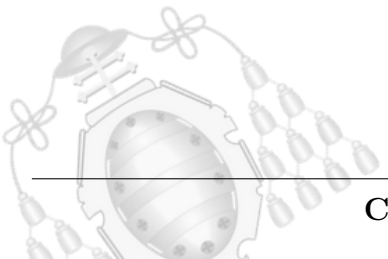
```
global config_file , network_ip, subnet_mask, gateway_ip, device_ip
a_yaml_file = open("config.yaml")
config_file = yaml.load(a_yaml_file, Loader=yaml.FullLoader)
network_ip = config_file[dpid]['PVLAN_MODULE']['Network_IP']
subnet_mask = config_file[dpid]['PVLAN_MODULE']['Subnet_Mask']
gateway_mac = config_file[dpid]['PVLAN_MODULE']['Gateway_MAC']
gateway_ip = config_file[dpid]['PVLAN_MODULE']['Gateway_IP']
def send_port_desc_stats_request(self, datapath):
ofp_parser = datapath.ofproto_parser
req = ofp_parser.OFPPortDescStatsRequest(datapath, 0)
datapath.send_msg(req)


@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
datapath = ev.msg.datapath
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
match = parser.OFPMatch()
actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
ofproto.OFPCML_NO_BUFFER)]
self.add_flow(datapath, 0, match, actions)
self.logger.debug('Features request successfully sent')
self.send_port_desc_stats_request(datapath)
def add_flow(self, datapath, priority, match, actions, buffer_id=None):
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
actions)]
if buffer_id:
mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
priority=priority, match=match,
instructions=inst)
else:
mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
match=match, instructions=inst)
datapath.send_msg(mod)
def add_flow_routing_protocols(self,datapath,dest):
ofproto = datapath.ofproto
```

```
parser = datapath.ofproto_parser
# IP Protos
# OSPF: 0x59
# EIGRP: 0x58
ip_protocols = (0x0059,0x0058)
match1 = parser.OFPMatch(in_port=port,eth_type=ip_protocols)
match2 = parser.OFPMatch(eth_type=ip_protocols)
actions1 = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER)]
actions2 = [parser.OFPInstructionActions(ofproto.OFPIT_CLEAR_ACTIONS,[])]
self.add_flow(datapath, 2, match1, actions1)
self.add_flow(datapath, 1, match2, actions2)
def drop_routing_traffic(self,datapath,proto,port):
parser = datapath.ofproto_parser
ofproto = datapath.ofproto
match = parser.OFPMatch(eth_type=ether_types.ETH_TYPE_IP,ip_proto=proto,
    in_port=port)
actions = [parser.OFPInstructionActions(ofproto.OFPIT_CLEAR_ACTIONS,[])]
mod = parser.OFPFlowMod(datapath=datapath,
out_port=ofproto.OFPP_ANY,
out_group=ofproto.OFPG_ANY,
match=match, instructions=actions)
#self.add_flow(datapath, 2, match, actions)
datapath.send_msg(mod)
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
#Aqui es donde debemos actuar con la vaina
def _packet_in_handler(self, ev):
# If you hit this you might want to increase
# the "miss_send_length" of your switch
global gateway_mac
global gateway_port
global gateway_port_numbers
global ospf_counter, eigrp_counter
if ev.msg.msg_len < ev.msg.total_len:
self.logger.debug("packet truncated: only %s of %s bytes",
ev.msg.msg_len, ev.msg.total_len)
msg = ev.msg
datapath = msg.datapath
dpid = format(datapath.id, "d").zfill(16)
```

```
self.update_yaml_file(dpid)
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
in_port = msg.match['in_port']
port_info = self.dpset.get_port(datapath.id, in_port)
hw_addr = port_info[1]
pkt = packet.Packet(msg.data)
eth = pkt.get_protocols(ethernet.ethernet)[0]
dst = eth.dst
src = eth.src
if eth.ethertype == ether_types.ETH_TYPE_LLDP:
# ignore lldp packet
return
if eth.ethertype == ether_types.ETH_TYPE_IP:
ip = pkt.get_protocol(ipv4.ipv4)
srcip = ip.src
dstip = ip.dst
proto = ip.proto
self.mac_to_port.setdefault(dpid, {})
self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)
if eth.ethertype == ether_types.ETH_TYPE_IP:
ip = pkt.get_protocol(ipv4.ipv4)
if (dst == gateway_mac) and (dstip != gateway_ip):
self.logger.debug('WARNING: Paquete con MAC destino Puerta de enlace fraudulento')
#Aqui debiera ir el envio de regla para dump en la tabla de matching
self.add_flow_pvlan(datapath,gateway_ip,subnet_mask)
self.logger.debug('INFO: Nueva regla anadida en el switch satisfactoriamente.')
#return
if proto == 88:
self.logger.debug("      ------------------------------      ")
self.logger.debug("EIGRP PACKET RECEIVED. PORT %s",in_port)
self.logger.debug("      ------------------------------      ")
if in_port not in config_file[dpid]['PROTOCOLS_MODULE']['Ports_enabled']:
if in_port not in eigrp_counter:
eigrp_counter[in_port]={'counter':1,'timestamp':current_milli_time()}
else:
deltha = int(current_milli_time()) - int(eigrp_counter[in_port]['timestamp'])
eigrp_counter[in_port]['timestamp'] = current_milli_time()
```

```
if deltha < 200:
eigrp_counter[in_port]['counter'] += 1
if (eigrp_counter[in_port]['counter']) >100:
self.logger.debug("WARNING: EIGRP FLOOD ATTACK DETECTED")
#Se enviaria o bien chapar el puerto o que se dropee el trafico
#self.shutdown_port(datapath,in_port,hw_addr)
self.drop_routing_traffic(datapath,0x0058,in_port)
return
self.logger.debug("WARNING: PORT NOT ALLOWED TO RECEIVE EIGRP PACKET.
    DROPPING")
return
if proto == 89:
self.logger.debug("    ------------------------------    ")
self.logger.debug("OSPF PACKET RECEIVED. PORT %s",in_port)
self.logger.debug("    ------------------------------    ")
if in_port not in config_file[dpid]['PROTOCOLS_MODULE']['Ports_enabled']:
if in_port not in ospf_counter:
ospf_counter[in_port]={'counter':1,'timestamp':current_milli_time()}
else:
deltha = int(current_milli_time()) - int(ospf_counter[in_port]['timestamp'])
ospf_counter[in_port]['timestamp'] = current_milli_time()
if deltha < 200:
ospf_counter[in_port]['counter'] += 1
if (ospf_counter[in_port]['counter']) >100:
self.logger.debug("WARNING: OSPF FLOOD ATTACK DETECTED")
#Se enviaria o bien chapar el puerto o que se dropee el trafico
#self.shutdown_port(datapath,in_port,hw_addr)
self.drop_routing_traffic(datapath,0x0059,in_port)
self.logger.debug("WARNING: PORT NOT ALLOWED TO RECEIVE OSPF PACKET.
    DROPPING")
return
if proto == 17:
udp_pkt = pkt.get_protocol(udp.udp)
if (udp_pkt.src_port == 520) or (udp_pkt.dst_port==520):
self.logger.debug("    ------------------------------    ")
self.logger.debug("RIP PACKET RECEIVED. PORT %s",in_port)
self.logger.debug("    ------------------------------    ")
if in_port not in config_file[dpid]['PROTOCOLS_MODULE']['Ports_enabled']:
```

```
self.logger.debug("WARNING: PORT NOT ALLOWED TO RECEIVE RIP PACKET.
    DROPPING")
return
if ((udp_pkt.src_port == 1985) or (udp_pkt.dst_port==1985)) and (dstip == "224.0.0.2"):
self.logger.debug("     ------------------------------     ")
self.logger.debug("HSRP PACKET RECEIVED. PORT %s",in_port)
self.logger.debug("     ------------------------------     ")
if in_port not in config_file[dpid]['PROTOCOLS_MODULE']['Ports_enabled']:
self.logger.debug("WARNING: PORT NOT ALLOWED TO RECEIVE HSRP PACKET.
    DROPPING")
return
self.mac_to_port[dpid][src] = in_port
if dst in self.mac_to_port[dpid]:
out_port = self.mac_to_port[dpid][dst]
else:
out_port = ofproto.OFPP_FLOOD
actions = [parser.OFPActionOutput(out_port)]
# install a flow to avoid packet_in next time
if out_port != ofproto.OFPP_FLOOD:
match = parser.OFPMatch(in_port=in_port, eth_dst=dst, eth_src=src)
# verify if we have a valid buffer_id, if yes avoid to send both
# flow_mod & packet_out
if msg.buffer_id != ofproto.OFP_NO_BUFFER:
self.add_flow(datapath, 1, match, actions, msg.buffer_id)
return
else:
self.add_flow(datapath, 1, match, actions)
data = None
if msg.buffer_id == ofproto.OFP_NO_BUFFER:
data = msg.data
out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
in_port=in_port, actions=actions, data=data)
datapath.send_msg(out)
```



Anexo V. Aplicación para detener ataques de doble tagging

Código 5.5.- Aplicación para detener ataques de doble tagging

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import vlan
from ryu.lib.packet import ether_types

#GLOBAL VARIABLES
port_vlan = {2855770673994:{10:[20],4:[20]},143080643644224:{2:[30],10:[20]}}
access= {2855770673994:[],143080643644224:[2]}
trunk = {2855770673994:[4,10],143080643644224:[10]}
security_enabled = True
class VlanSwitch13(app_manager.RyuApp):
OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
def __init__(self, *args, **kwargs):
super(VlanSwitch13, self).__init__(*args, **kwargs)
self.mac_to_port = {}
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
datapath = ev.msg.datapath
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
match = parser.OFPMatch()
actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
ofproto.OFPCML_NO_BUFFER)]
self.add_flow(datapath, 0, match, actions)
def add_flow(self, datapath, priority, match, actions, buffer_id=None):
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
actions)]
```

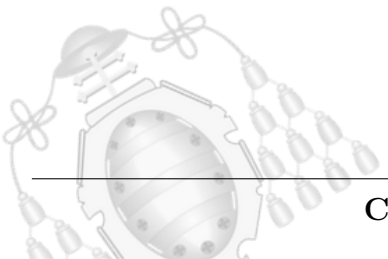
```
if buffer_id:
mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
priority=priority, match=match,
instructions=inst)
else:
mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
match=match, instructions=inst)
datapath.send_msg(mod)
def vlan_members(self,dpid,in_port,src_vlan):
B=[]
self.access_ports = []
self.trunk_ports = []
if src_vlan == "NULL":
return
#Mira que puertos tienen esa VLAN exceptuando el que lo recibio
for item in port_vlan[dpid]:
vlans=port_vlan[dpid][item]
if src_vlan in vlans and item!=in_port:
B.append(item)
#De esos puertos, mira los que son de acceso y los que son troncales
for port in B:
if port in access[dpid]:
self.access_ports.append(port)
else:
self.trunk_ports.append(port)
#          ------------------------------------------------------------#
def getActionsArrayTrunk(self,out_port_access,out_port_trunk,parser):
actions= [ ]
for port in out_port_trunk:
actions.append(parser.OFPActionOutput(port))
actions.append(parser.OFPActionPopVlan())
for port in out_port_access:
actions.append(parser.OFPActionOutput(port))
return actions
def getActionsArrayTrunkNoNativa(self,out_port_access,out_port_trunk,parser,native_vlan):
actions= [ ]
#Si no esta etiquetada la nativa, tenemos que forzar el asunto
actions.append(parser.OFPActionPushVlan(33024))
```

```
actions.append(parser.OFPActionSetField(vlan_vid=native_vlan))
for port in out_port_trunk:
actions.append(parser.OFPActionOutput(port))
for port in out_port_access:
actions.append(parser.OFPActionOutput(port))
return actions
def getActionsArrayAccess(self,out_port_access,out_port_trunk,src_vlan, parser):
actions= [ ]
for port in out_port_access:
actions.append(parser.OFPActionOutput(port))
actions.append(parser.OFPActionPushVlan(33024))
actions.append(parser.OFPActionSetField(vlan_vid=src_vlan))
for port in out_port_trunk:
actions.append(parser.OFPActionOutput(port))
return actions
def getActionsNormalUntagged(self,dpid,in_port,parser):
actions= [ ]
for port in port_vlan[dpid]:
if port_vlan[dpid][port][0]==" " and port!=in_port:
actions.append(parser.OFPActionOutput(port))
if dpid in trunk:
for port in trunk[dpid]:
if port != in_port:
actions.append(parser.OFPActionOutput(port))
return actions
def check_out_port_trunk(self,dpid,out_port_trunk,src_vlan):
checker = [False,""]
for port in out_port_trunk:
if src_vlan in port_vlan[dpid][port]:
checker = [True,port_vlan[dpid][port]]
return checker
checker[1]=port_vlan[dpid][port]
return checker
#          ------------------------------------------------------------#
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
# If you hit this you might want to increase
# the "miss_send_length" of your switch
```

```
if ev.msg.msg_len < ev.msg.total_len:
self.logger.debug("packet truncated: only %s of %s bytes",
ev.msg.msg_len, ev.msg.total_len)
msg = ev.msg
datapath = msg.datapath
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
in_port = msg.match['in_port']
dpid = datapath.id
pkt = packet.Packet(msg.data)
eth = pkt.get_protocols(ethernet.ethernet)[0]
vlan_header = pkt.get_protocols(vlan.vlan)
if eth.ethertype == ether_types.ETH_TYPE_8021Q :
vlan_header_present = 1
src_vlan=vlan_header[0].vid
self.logger.debug("\n802.1Q PACKET RECEIVED. SRC VLAN: %s",src_vlan)
elif dpid not in port_vlan:
vlan_header_present = 0
in_port_type = "NORMAL SWITCH "
src_vlan = "NULL"
elif port_vlan[dpid][in_port][0]== " " or in_port in trunk[dpid]:
vlan_header_present = 0
in_port_type = "NORMAL UNTAGGED"
src_vlan = "NULL"
self.logger.debug("NORMAL UNTAGGED PACKET")
else:
vlan_header_present = 0
src_vlan=port_vlan[dpid][in_port][0]
if eth.ethertype == ether_types.ETH_TYPE_LLDP:
return
dst = eth.dst
src = eth.src
self.mac_to_port.setdefault(dpid, {})
self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)
self.mac_to_port[dpid][src] = in_port
self.vlan_members(dpid,in_port,src_vlan)
out_port_type = " "
if dst in self.mac_to_port[dpid]:
```

```
out_port_unknown = 0
out_port = self.mac_to_port[dpid][dst]
if src_vlan!= "NULL":
if out_port in access[dpid]:
out_port_type = "ACCESS"
else:
out_port_type = "TRUNK"
else :
out_port_type = "NORMAL"
else:
out_port_unknown = 1
out_port_access = self.access_ports
out_port_trunk = self.trunk_ports
self.logger.debug("DEST: %s. OUT PORT TYPE: %s \n",dst,out_port_type)
if out_port_unknown!=1:
if vlan_header_present and out_port_type == "ACCESS":
match = parser.OFPMatch(in_port=in_port, eth_dst=dst, vlan_vid=(0x1000 | src_vlan))
actions = [parser.OFPActionPopVlan(), parser.OFPActionOutput(out_port)]
elif vlan_header_present and out_port_type == "TRUNK":
match = parser.OFPMatch(in_port=in_port, eth_dst=dst, vlan_vid=(0x1000 | src_vlan))
actions = [parser.OFPActionOutput(out_port)]
elif vlan_header_present!=1 and out_port_type == "TRUNK":
self.logger.debug("OUT PORT IS TRUNK AND PACKET HAS NO HEADER")
match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
actions = [parser.OFPActionPushVlan(33024), parser.OFPActionSetField(vlan_vid=src_vlan),
    parser.OFPActionOutput(out_port)]
else:
match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
actions = [parser.OFPActionOutput(out_port)]
if msg.buffer_id != ofproto.OFP_NO_BUFFER:
self.add_flow(datapath, 1, match, actions, msg.buffer_id)
return
else:
self.add_flow(datapath, 1, match, actions)
else :
if vlan_header_present:
#Aqui tengo que ver si la VLAN de la primera etiqueta coincide con la de la nativa
if security_enabled:
```

```
if out_port_trunk != ['']:
passer=self.check_out_port_trunk(dpid,out_port_trunk,src_vlan)
if passer[0]:
#self.logger.debug("LA VLAN NATIVA ESTA PRESENTE EN LA PRIMERA ETIQUETA:
    %s",src_vlan)
actions = self.getActionsArrayTrunk(out_port_access,out_port_trunk,parser)
match = parser.OFPMatch(in_port=in_port, eth_dst=dst, vlan_vid=(0x1000 | src_vlan))
#self.add_flow(datapath, 1, match, actions)
else:
#self.logger.debug("LA VLAN NATIVA NO ESTA PRESENTE EN LA PRIMERA
    ETIQUETA: %s",src_vlan)
match = parser.OFPMatch(in_port=in_port, eth_dst=dst, vlan_vid=(0x1000 | src_vlan))
actions =
    self.getActionsArrayTrunkNoNativa(out_port_access,out_port_trunk,parser,passer[1])
#self.add_flow(datapath, 1, match, actions)
if out_port_access != ['']:
actions = self.getActionsArrayTrunk(out_port_access,out_port_trunk,parser)
else:
actions = self.getActionsArrayTrunk(out_port_access,out_port_trunk,parser)
match = parser.OFPMatch(in_port=in_port, eth_dst=dst, vlan_vid=(0x1000 | src_vlan))
elif vlan_header_present==0 and src_vlan!= "NULL": #IF UNTAGGED BUT
    GENERATED FROM VLAN ASSOCIATED PORT
actions = self.getActionsArrayAccess(out_port_access,out_port_trunk,src_vlan, parser)
elif in_port_type == "NORMAL UNTAGGED":
actions = self.getActionsNormalUntagged(dpid,in_port,parser)
else :
actions = [parser.OFPActionOutput(ofproto.OFPP_FLOOD)]
data = None
if msg.buffer_id == ofproto.OFP_NO_BUFFER:
data = msg.data
out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
in_port=in_port, actions=actions, data=data)
datapath.send_msg(out)
```



Anexo VI. Descubrimiento de equipos en red mediante protocolo LLDP

En la imagen 5.1 se muestra un escenario tipo, formado por cuatro switches conectados entre sí (S1,S2,S3 y S4), tres ordenadores (H2,H3 y H4) y un controlador (H1).

Una vez se establecen las conexiones entre los switches de la red, el controlador les manda la orden de que reenvíen por todos sus puertos los paquetes LLDP. A continuación, un paquete LLDP es enviado al switch más cercano (S1) que le llega por el puerto número 3. S1 reenvía este paquete a todos los switches conectados a el, que en este escenario de ejemplo serán S2 y S3. El paquete reenviado por S1 en el puerto 1 llegará a S3 en el puerto 1, y el paquete reenviado por S1 en el puerto 2 llega a S2 por el puerto 2. Una vez lleguen los paquetes a S2 y S3, estos reenvían los paquetes LLDP por todos sus puertos, llegando a S4 en este caso tanto por el puerto 1 como por el puerto 2.

A parte de reenviar los paquetes LLDP por cada puerto, cada switch envía una respuesta al controlador. Como en cada respuesta van acumulándose los saltos que realiza cada paquete, una vez lleguen al controlador, este sabe que S3 está conectado a S1 a través de su puerto número 1 y que S2 está conectado a S1 a través de su puerto número 2. De manera análoga se realiza con S4 y con cuantos niveles de switches conectados se quiera. De esta forma el controlador es capaz de recrear toda la topología de la red.

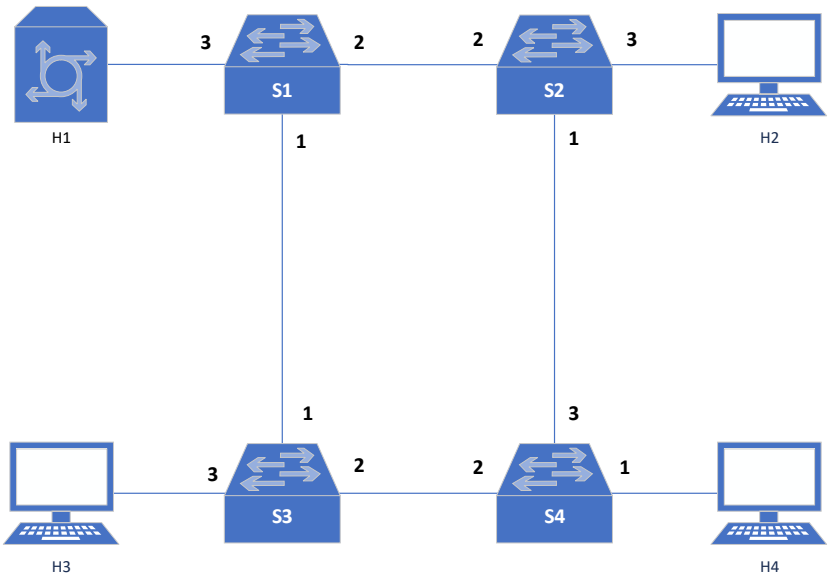


Figura 5.1.- Escenario de una red ejemplo conformada por 4 switches, 3 ordenadores y un controlador.

Anexo VII. Aplicación base de Ryu con funcionalidades VLAN

Código 5.6.- Aplicación base de Ryu con funcionalidades VLAN

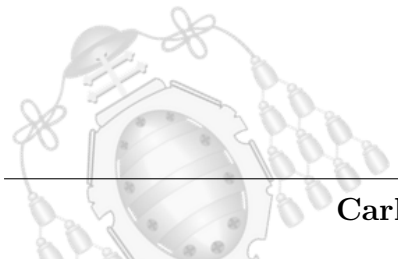
```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import vlan
from ryu.lib.packet import ether_types

#GLOBAL VARIABLES
port_vlan = {2:{3:[" "      ],1:[30],2:[20],4:[20,30]},3:{4:[20,30],1:[30],2:[20],3:[      " "]}}
access=  {2:[1,2],3:[1,2]}
trunk =  {2:[4],3:[4]}
class VlanSwitch13(app_manager.RyuApp):
OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
def __init__(self, *args, **kwargs):
super(VlanSwitch13, self).__init__(*args, **kwargs)
self.mac_to_port = {}
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
datapath = ev.msg.datapath
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
match = parser.OFPMatch()
actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
ofproto.OFPCML_NO_BUFFER)]
self.add_flow(datapath, 0, match, actions)
def add_flow(self, datapath, priority, match, actions, buffer_id=None):
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
actions)]
if buffer_id:
```

```
mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
priority=priority, match=match,
instructions=inst)
else:
mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
match=match, instructions=inst)
datapath.send_msg(mod)
def vlan_members(self,dpid,in_port,src_vlan):
B=[]
self.access_ports = []
self.trunk_ports = []
if src_vlan == "NULL":
return
for item in port_vlan[dpid]:
vlans=port_vlan[dpid][item]
if src_vlan in vlans and item!=in_port:
B.append(item)
for port in B:
if port in access[dpid]:
self.access_ports.append(port)
else:
self.trunk_ports.append(port)
#         ------------------------------------------------------------#
def getActionsArrayTrunk(self,out_port_access,out_port_trunk,parser):
actions= [ ]
for port in out_port_trunk:
actions.append(parser.OFPActionOutput(port))
actions.append(parser.OFPActionPopVlan())
for port in out_port_access:
actions.append(parser.OFPActionOutput(port))
return actions
def getActionsArrayAccess(self,out_port_access,out_port_trunk,src_vlan, parser):
actions= [ ]
for port in out_port_access:
actions.append(parser.OFPActionOutput(port))
actions.append(parser.OFPActionPushVlan(33024))
actions.append(parser.OFPActionSetField(vlan_vid=src_vlan))
for port in out_port_trunk:
```

```
actions.append(parser.OFPActionOutput(port))
return actions
def getActionsNormalUntagged(self,dpid,in_port,parser):
actions= [ ]
for port in port_vlan[dpid]:
if port_vlan[dpid][port][0]==" " and port!=in_port:
actions.append(parser.OFPActionOutput(port))
if dpid in trunk:
for port in trunk[dpid]:
if port!=in_port:
actions.append(parser.OFPActionOutput(port))
return actions
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
# If you hit this you might want to increase
# the "miss_send_length" of your switch
if ev.msg.msg_len < ev.msg.total_len:
self.logger.debug("packet truncated: only %s of %s bytes",
ev.msg.msg_len, ev.msg.total_len)
msg = ev.msg
datapath = msg.datapath
ofproto = datapath.ofproto
parser = datapath.ofproto_parser
in_port = msg.match['in_port']
#SWITCH ID
dpid = datapath.id
pkt = packet.Packet(msg.data)
eth = pkt.get_protocols(ethernet.ethernet)[0]
vlan_header = pkt.get_protocols(vlan.vlan)   #If packet is tagged,then this will have non-null
    value
if eth.ethertype == ether_types.ETH_TYPE_8021Q : #Checking for VLAN Tagged Packet
vlan_header_present = 1
src_vlan=vlan_header[0].vid
elif dpid not in port_vlan:
vlan_header_present = 0
in_port_type = "NORMAL SWITCH "
src_vlan = "NULL"
elif port_vlan[dpid][in_port][0]== " " or in_port in trunk[dpid]:
```

```
vlan_header_present = 0
in_port_type = "NORMAL UNTAGGED"
src_vlan = "NULL"
else:
vlan_header_present = 0
src_vlan=port_vlan[dpid][in_port][0]
if eth.ethertype == ether_types.ETH_TYPE_LLDP:
# ignore lldp packet
return
dst = eth.dst
src = eth.src
#CREATE NEW DICTIONARY ENTRY IF IT DOES NOT EXIST
self.mac_to_port.setdefault(dpid, {})
self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)
# learn a mac address to avoid FLOOD next time.
self.mac_to_port[dpid][src] = in_port
#Determine which ports are members of in_port's VLAN
self.vlan_members(dpid,in_port,src_vlan)
out_port_type = " "
if dst in self.mac_to_port[dpid]:
out_port_unknown = 0
out_port = self.mac_to_port[dpid][dst]
if src_vlan!= "NULL":
if out_port in access[dpid]:
out_port_type = "ACCESS"
else:
out_port_type = "TRUNK"
else :
out_port_type = "NORMAL"
else:
out_port_unknown = 1
out_port_access = self.access_ports
out_port_trunk = self.trunk_ports
if out_port_unknown!=1:
if vlan_header_present and out_port_type == "ACCESS" :
match = parser.OFPMatch(in_port=in_port, eth_dst=dst, vlan_vid=(0x1000 | src_vlan))
actions = [parser.OFPActionPopVlan(), parser.OFPActionOutput(out_port)]
elif vlan_header_present and out_port_type == "TRUNK" :
```

```
match = parser.OFPMatch(in_port=in_port, eth_dst=dst, vlan_vid=(0x1000 | src_vlan))
actions = [parser.OFPActionOutput(out_port)]
elif vlan_header_present!=1 and out_port_type == "TRUNK" :
match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
actions = [parser.OFPActionPushVlan(33024), parser. OFPActionSetField(vlan_vid=src_vlan),
    parser.OFPActionOutput(out_port)]
else:
match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
actions = [parser.OFPActionOutput(out_port)]
# verify if we have a valid buffer_id, if avoid yes to send both
# flow_mod & packet_out
if msg.buffer_id != ofproto.OFP_NO_BUFFER:
self.add_flow(datapath, 1, match, actions, msg.buffer_id)
return
else:
self.add_flow(datapath, 1, match, actions)
else :
if vlan_header_present:
actions = self.getActionsArrayTrunk(out_port_access, out_port_trunk, parser)
elif vlan_header_present==0 and src_vlan!= "NULL":
actions = self.getActionsArrayAccess(out_port_access, out_port_trunk, src_vlan, parser)
elif in_port_type == "NORMAL UNTAGGED":
actions = self.getActionsNormalUntagged(dpid,in_port,parser)
else :
actions = [parser.OFPActionOutput(ofproto.OFPP_FLOOD)]
data = None
if msg.buffer_id == ofproto.OFP_NO_BUFFER:
data = msg.data
out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
in_port=in_port, actions=actions, data=data)
datapath.send_msg(out)
```



Bibliografía

[1] Diego Kreutz, Fernando MV Ramos, P Esteves Verissimo, C Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. *Software-defined networking: A comprehensive survey*
https://ieeexplore.ieee.org/document/6994333/

[2] Open Networking Foundation, *SDN Architecture*
https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/SDN-architecture-overview-1.0.pdf

[3] Rachid Nait Takourout, Samuel Pierre, Laurent Marchand, *Separation of the Control Plane and Forwarding Plane in Next-Generation Routers*
http://www.larim.polymtl.ca/pdf/52.pdf

[4] Jianying Luo, Justin Pettit, Martin Casado, John Lockwood, Nick McKeown, *Prototyping Fast, Simple, Secure Switches for Ethane*
http://yuba.stanford.edu/~nickm/papers/ethane-hoti07.pdf

[5] Andreas Wundsam, Dan Levin, Srini Seetharaman, Anja Feldmann, *OFRewind: Enabling Record and Replay Troubleshooting for Networks*

[6] SFlow
https://sflow-rt.com/index.php

[7] Nuño Huergo, Pelayo *Administración de Redes y Servicios - Software defined networks* (2019)

[8] OpenFlow. *OpenFlow Switch Specification version 1.0.0*
https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf

[9] OpenFlow. *OpenFlow Switch Specification version 1.1.0*
https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.1.0.pdf

[10] OpenFlow. *OpenFlow Switch Specification version 1.2.0* https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.2.pdf

[11] OpenFlow. *OpenFlow Switch Specification version 1.3.0* http://www.cs.yale.edu/homes/yu-minlan/teach/csci599-fall12/papers/openflow-spec-v1.3.0.pdf

[12] OpenFlow. *OpenFlow Switch Specification version 1.5.0* http://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.0.noipr.pdf

[13] Vivek Tiwari, (2013), *SDN and OpenFlow for begginers* (e-Book)

[14] Kevin Benton, L. Jean Camp, Chris Small, (2013), OpenFlow Vulnerability Assesment http://www.ljean.com/files/vulnerability_analysis.pdf

[15] Andi Bidaj, *Security Testing SDN Controllers* https://aaltodoc.aalto.fi/bitstream/handle/123456789/21584/master_Bidaj_Andi_2016.pdf(30 junio 2016)

[16] Daniel Romão, Niels van Dijkhuizen, Stavros Konstantaras, George Thessalonikefs, *Practical Security Analysis of OpenFlow* https://www.os3.nl/_media/2013-2014/courses/ssn/projects/practical_security_analysis_of_openflow_report.pdf(24 Diciembre 2013)

[17] Sumanth M. Sathyanarayana, *Software Defined Networks Defense* http://www.academia.edu/1833986/Software_defined_network_defense (2012)

[18] SDX Central, *Can OpenFlow Scale to Build Large Networks?* https://www.sdxcentral.com/articles/contributed/openflow-sdn/2013/06/ (2013)

[19] Flowgrammable.org, *OpenFlow. Message layer* http://flowgrammable.org/sdn/openflow/message-layer/

[20] Huawei, *Network Management and Monitoring*
https://support.huawei.com/enterprise/en/doc/EDOC1000150290/c824b277/openflow-working-mechanism

[21] Tiantian Ren, Yanwei Xu, *Analysis of the new features of the OpenFlow 1.4*
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.827.7016&rep=rep1&type=pdf (2014)

[22] Rowan Klöti, Vasileios Kotronis, Paul Smith, *OpenFlow: A Security Analysis*
https://www.tik.ee.ethz.ch/file/9ee69e89be779fd1448a7356d79ddb18/openflow_sec.pdf (2013)

[23] Ching-Hao, Chang and Dr. Ying-Dar Lin, *OpenFlow Version Roadmap*
http://speed.cis.nctu.edu.tw/~ydlin/miscpub/indep_frank.pdf
(Septiembre 2015)

[24] M. Ambrosin, M. Conti, F. De Gaspari, R. Poovendran, *LineSwitch: Tackling Control Plane Saturation Attacks in Software-Defined Networking*
https://www2.ee.washington.edu/research/nsl/papers/TON.pdf

[25] Jeremy M. Dover, *A denial of service attack against the Open Floodlight SDN controller*
http://dovernetworks.com/wp-content/uploads/2013/12/OpenFloodlight-12302013.pdf

[26] Gregory Pickett, *Abusing Software Defined Networks*
https://www.blackhat.com/docs/eu-14/materials/eu-14-Pickett-Abusing-Software-Defined-Networks-wp.pdf (2014)
http://ecee.colorado.edu/~ekeller/classes/spring2014_advnet/papers/ofrewind_usenixatc_2011_wendsam.pdf

[27] Seungwon Shin, Vinod Yegneswaran, Phillip Porras, Guofei Gu, *AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks* (2013)

[28] Jeremy M. Dover *A switch table vulnerability in the FloodLight controller* http://dovernetworks.com/wp-content/uploads/2014/03/OpenFloodlight-03052014.pdf

[29] Seclists.org, Kashyap Thimmaraju, *Denial of Service, Improper Authentication and Authorization, and Covert Channel in the OpenFlow 1.0+ handshake* http://seclists.org/oss-sec/2018/q2/99

[30] Nuño Huergo, Pelayo *Administración de Redes y Servicios - OpenFlow* (2019)

[31] Putty.org https://www.putty.org/

[32] Wikipedia.org , *Xming Server* https://es.wikipedia.org/wiki/Xming

[33] Nmap.org https://nmap.org/

[34] Readthedocs.org, *Getting started - Ryu installation* https://ryu.readthedocs.io/en/latest/getting_started.html

[35] OpenDayLight Git https://git.opendaylight.org/

[36] NAKIVO, *Hyper-V or VirtualBox. Which one to choose for your infrastructure?* https://www.nakivo.com/blog/hyper-v-virtualbox-one-choose-infrastructure/

[37] VMWare, *ESXi* https://www.vmware.com/es/products/esxi-and-esx.html

[38] VirtualBox *VirtualBox webpage* https://www.virtualbox.org/

[39] Open Networking Foundation *ONOS* https://www.opennetworking.org/onos/

[40] The New Stack, *NOX, the original OpenFlow Controller*
https://thenewstack.io/sdn-series-part-iii-nox-the-original-openflow-controller/

[41] POX Documentation, *GitHub*
https://noxrepo.github.io/pox-doc/html/

[42] Mininet, *An Instant Virtual Network on your Laptop*
http://mininet.org/

[43] Murphy MC, *POX Web GUI*
https://github.com/MurphyMc/poxdesk/wiki/Getting-Started

[44] PyPy, *A fast, compliant alternative implementation of Python*
https://www.pypy.org/

[45] NSCR.org *Ryu, a SDN Controller*
https://nsrc.org/workshops/2014/nznog-sdn/raw-attachment/wiki/WikiStart/Ryu.pdf

[46] RFC 5517, *Private VLAN*
https://tools.ietf.org/html/rfc5517

[47] What is Open vSwitch? *Open vSwitch website*
https://www.openvswitch.org/

[48] GNS3 Docs, *OpenVSwitch*
https://docs.gns3.com/appliances/openvswitch.html

[49] Wikipedia.org, *Open vSwitch*
https://en.wikipedia.org/wiki/Open_vSwitch

[50] Rubén Usamentiaga Fernández, *Sistemas distribuidos: Servicios web* (2017)

[51] Oluwadamilola Adenuga-Taiwo, Shahra Shah Heydari, *Security analysis of ONOS software-defined network platform*
https://ir.library.dc-uoit.ca/bitstream/10155/685/1/ONOS%20Security%20TECHNICAL%20REPORT.pdf (2016)