TRABAJO FIN DE GRADO

INGENIERÍA EN INFORMÁTICA

Implementación en tiempo real de sistemas de identificación de tráfico de red

Subtitulo del proyecto

Autor
Álvaro Maximino Linares Herrera

Directores
Jesús Esteban Díaz Verdejo

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, 3 de septiembre de 2017

Implementación en tiempo real de sistemas de identificación de tráfico de red

Subtitulo del proyecto

Autor
Álvaro Maximino Linares Herrera

Directores
Jesús Esteban Díaz Verdejo

Implementación en tiempo real de sistemas de identificación de tráfico de red

Álvaro Maximino Linares Herrera

Palabras clave: inspección profunda de paquetes, bro, flujos, identificación, clasificación, red, tráfico

Resumen

La identificación de tráfico en red es realmente importante para aplicaciones de ingeniería de tráfico y de seguridad.

En este trabajo se tratará la creación de un programa para un NMS (Network Monitoring System), en este caso se usará BRO, mediante el cual se pueda resolver el emparejamiento de flujos. BRO consiste en un NMS que funciona mediante el terminal en Linux o Mac, una de las peculiaridades de este programa es que para la creación de scripts que nos permitan extender la funcionalidad de la que dispone, tendremos que usar BRO como lenguaje de programación. Es un lenguaje de scripting, el cual está orientado a eventos, que se lanzan cuando ocurre algo relacionado con el control y análisis de redes, es un lenguaje que para los que vienen de C++, Java o Python, no debe de suponer un gran reto, más allá de acostumbrarse a sus sintaxis. Es un lenguaje potente que al estar orientado a redes nos permite obtener mucha información de los flujos que tenemos en la red o en el archivo que vayamos a analizar. En este trabajo mediante implementaciones offline se verificará la eficacia de esta técnica de clasificación de tráfico.

Project Title: Project Subtitle

First name, Family name (student)

Keywords: Keyword1, Keyword2, Keyword3, ...

Abstract

Write here the abstract in English.

Yo, **Álvaro Maximino Linares Herrera**, alumno de la titulación TITULACIÓN de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 76669401M, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Álvaro Maximino Linares Herrera

Granada a 3 de septiembre de 2017.

D. **Jesús Esteban Díaz Verdejo**, Profesor del Área de XXXX del Departamento YYYY de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado ***Implementación en tiempo real de sistemas de identificación de tráfico de red***, ha sido realizado bajo su supervisión por **Álvaro Maximino Linares Herrera**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 3 de septiembre de 2017.

Los directores:

Jesús Esteban Díaz Verdejo **Nombre Apellido1 Apellido2 (tutor2)**

Agradecimientos

Poner aquí agradecimientos...

Índice general

Capítulo 1

Introducción

La clasificación de tráfico en red es una tarea importante en lo relativo a las comunicaciones, en un mundo cada vez más digitalizado e intercomunicado, lo que más importa es la seguridad y para ello es esencial la clasificación del tráfico, permitiendo detectar de forma temprana intrusiones y comportamientos anómalos, por ejemplo podríamos ver, mediante la clasificación del tráfico si estamos siendo atacados mediante denegación de servicio, *DDoS*, y podríamos tomar medidas para cortar esa conexión. De esta forma, por ejemplo, el encargado de un servidor podrá mantener la calidad de servicio, pues mediante *ISP (Internet Service Provider)* se puede establecer diferentes niveles de prioridad en el tráfico de red.

En este trabajo haremos uso de BRO, un NMS (Network Monitoring System), al cual mediante la implementación de técnicas de emparejamiento de flujos, lo dotaremos de la capacidad analítica de discernir que flujos son emparejables, y por lo tanto pertenecen a la misma conexión.

1.1. Capas de Red

Como ya sabrá, Internet sigue un modelo capas y que hay varios modelos para organizar estas capas, como el TCP/IP y el OSI [13].

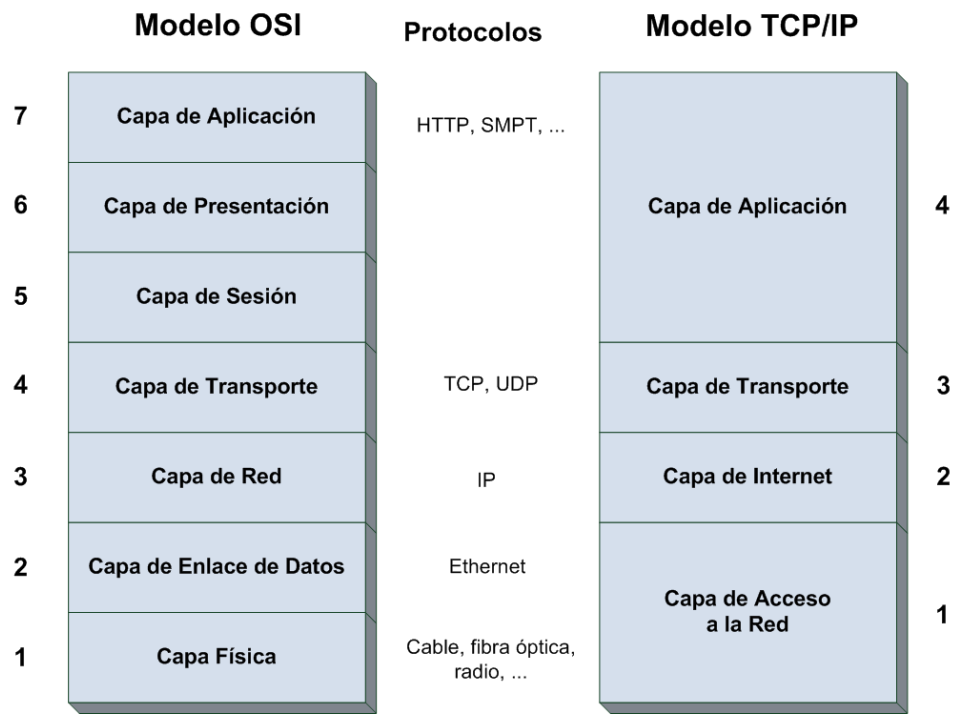

Figura 1.1: Modelos de capas de Internet.

Nosotros seguiremos el *modelo TCP/IP*, y a continuación, procederemos a dar unas pinceladas sobre la capa de aplicación, la de transporte y la de enlace.

1.1.1. Capa de aplicación

En la capa de aplicación encontramos las aplicaciones de red y sus protocolos. Los protocolos que encontramos en esta capa son **HTTP, SMTP y FTP**.

- El protocolo *HTTP* permite la solicitud y transferencia de documentos web.
- El protocolo *SMTP* permite la transferencia de mensajes de correo electrónico.
- El protocolo *FTP* permite la transferencia de archivos entre dos terminales.

A parte de estos protocolos también corresponde a esta capa el **DNS**, que es el encargado de traducir los nombres de los sitios web.

1.1.2. Capa de transporte

La capa de transporte es la encargada de transportar los mensajes de la capa de aplicación. Los protocolos de esta capa son **TCP y UDP**. La principal diferencia y será algo que marque el trabajo es que *TCP* está orientado a garantizar la conexión, mientras que *UDP* no garantiza la conexión.

1.1.3. Capa de red

En la capa de red es donde nos encontramos el protocolo **IP** y también un protocolo, **ICMP**, que nuestro *NMS* hay veces que localiza y sitúa junto con *TCP y UDP*, aunque no pertenezca a la capa de transporte.

En esencia, esta capa es la que se encarga de transportar los *datagramas* de un *host* a otro.

1.1.4. ¿Qué es útil para nosotros?

En nuestro caso nos quedaremos solamente en la capa de transporte, ignorando la información que podamos obtener de la capa de aplicación. En caso de que nuestro NMS lo detecte, incluiremos ICMP, pero solo para añadir más información a los distintos flujos que podemos llegar a analizar, pero despreciaremos la información relativa al resto de esta capa. El archivo pcap que utilizaremos para la prueba será *nitroba.pcap*, el cual se puede obtener desde la web de Bro y el cual se corresponde con la obtención del tráfico de una universidad de Sudáfrica.

Capítulo 2

Estado del arte

La idea de la que partimos es la siguiente: Los flujos son un conjunto de paquetes que comparten la misma información, IP de origen, IP de destino y los puertos de origen y destino, aparte de esta información también tienen un tiempo de inicio, relativo al *timestamp* del primer paquete del flujo y una duración. Con estos datos ya podemos empezar a trabajar en el emparejamiento, pues si dos flujos comparten las IP's de origen y destino y los puertos de origen y destino ya serían candidatos a la comparación.

Para comparar los flujos lo que haremos será aplicar la siguiente fórmula:

$$F(x,y) = \begin{cases} G(x,y), & NIP(x,y) \geq 1 \\ -\infty, & \text{en otro caso} \end{cases}$$

Donde tenemos que G(x,y) es:

$$G(x,y) = |NIP(x,y)–1|+1/(dp1(x,y)+k1)+1/(dp2(x,y)+k1)+1/(dt(x,y)+k2)$$

Tenemos que **x** e **y** son los flujos a comparar, **NIP(x,y)** es el número de veces que tenemos almacenado flujos similares, por lo cual siempre será uno, como mínimo. **dp1(x,y)** son los puertos de origen de los dos flujos, que al comparar siempre flujos que tienen los mismos puertos serán siempre el mismo puerto. **dp2(x,y)** son los puertos de destino, que como en el caso anterior siempre serán el mismo puerto. **k1** y **k2** son constantes definidas antes de ejecutar el programa, estando estas normalmente en los trabajos realizados anteriormente estas constantes suelen estar definidas entre 1 y 10000 y, por último, **dt** es la diferencia de tiempo entre los **timestamps** de los dos flujos.

Con esta fórmula obtendremos un número que podremos comparar con un umbral que estará definido a la hora de ejecutar el programa, si el umbral definido es pequeño tendremos más flujos almacenados, si el umbral es más grande serán menos los flujos que serán emparejables y por tanto almacenados para trabajar con ellos. [1]

2.1. DPI

La Inspección Profunda de Paquetes o DPI, por sus siglas en inglés, Deep Packet Inspection, consiste en analizar un paquete entero, no solo la cabecera del paquete. En la cabecera de un paquete suele ir el protocolo, el tiempo de vida del paquete y las direcciones IP's de origen y destino. Como podrá observar a nosotros también nos interesan los puertos de origen y destino para poder realizar nuestra comprobación mediante la fórmula expuesta anteriormente. Gracias a esta técnica podemos detectar intrusiones, ataques de denegación de servicio y demás [2], lo cual es bastante interesante, ya que el administrador podrá saber qué medidas tomar respecto a las intrusiones. Este análisis también sirve para realizar estadísticas sobre el tráfico de una red, que en nuestro caso nos interesa bastante, pues nosotros vamos a realizar emparejamiento de flujos, mediante la información de sus paquetes, siendo el tiempo del primer paquete una de las medidas que usamos como variable para la comparación de los flujos. Aunque esta técnica es más costosa que la inspección de paquetes poco profunda, la cual consiste en analizar solamente las cabeceras de los paquetes, para nosotros es necesaria, pues necesitamos saber los puertos.

2.2. NMS y BRO

Bro es un NMS, Network Monitoring System, cuya principal actividad es analizar el tráfico de una red para la búsqueda de amenazas. Una de las muchas peculiaridades de Bro es que usa un lenguaje propio, llamado también Bro, el cual tiene similitudes con *Python* y *Java*, y se gestiona mediante **eventos**. En el caso de este trabajo se utilizará dicho lenguaje, el cual en un principio puede parecer difícil, pues no disponemos de más documentación que la propia de la web de Bro [3], pero que acaba siendo bastante útil para la tarea que queremos, pues tiene un tipo de dato llamado *connection*, el cual está relacionado con los flujos, siendo al final este tipo el que representa al propio flujo. En este tipo se establecen bastantes datos que nos serán útiles, como la IP de origen, la de destino, así como los puertos y sus protocolos, también guarda la información de la duración así como el *timestamp* y más información que, en este caso, no será relevante, como la *historia del flujo*, donde se nos informa del tipo de paquetes que contiene e incluso el *número de paquetes* de los que consta el flujo.

En el caso de los eventos, también nos facilitarán el trabajo, pues tenemos eventos que gestionan todo lo que tiene que ver con los flujos, desde cuando aparece un nuevo flujo el cual no tenemos identificado previamente, lo cual para nuestro trabajo es fundamental, pues nos hace el programa más eficiente, ya que cada vez que se lanza ese evento significa que tenemos que añadir ese flujo a la lista de flujos comparables, sin tener que compararlo con los que tengamos previamente almacenados. También tenemos flujos que nos indicarán cuando un flujo esté a punto de morir, siendo el momento de eliminarlo de la lista de comparables y, en caso de tener flujos con los que éste ha sido comparado, pasará el primero de estos a reemplazar el flujo que va a morir en la lista de comparables, de ésta forma nos evitamos tener que estar comprobando la duración del flujo para eliminarlo manualmente. Con todo ésto conseguimos un programa eficiente en cuanto a tiempo y que, gracias a estos eventos, va a identificar los dos protocolos de la capa de transporte, siendo estos **TCP** y **UDP**. Cabe destacar que los eventos de *UDP* empleados en nuestro programa son muy costoso, por lo tanto en caso de tener mucho tráfico *UDP*, siendo de dos tipos distinto **reply** y **request**, puede hacer que el análisis tarde más tiempo del deseado, pero el programa suele funcionar en una horquilla de tiempo bastante aceptable en términos de rendimiento para el análisis del tráfico, aunque cabe destacar que en nuestro caso analizamos archivos, no tráfico en tiempo real. En el apartado *Implementación* se hablará sobre los eventos usados con más detalle.

Bro también nos otorga una gran cantidad de **logs** en los cuales él mismo puede llegar a clasificar el tráfico dependiendo del protocolo que se use, siendo algunos ejemplos, *HTTP, DNS, SSL, etc.* pero no nos interesa, pues se tratan de protocolos de la capa de aplicación. Aunque como en nuestro caso usamos la información contenida en un archivo **pcap**, la información de los protocolos de la capa de aplicación, dependerá de si han sido cargados los *scripts* correspondientes cuando se realizó la captura del tráfico, pues en los flujos de Bro la información relativa a *HTTP, DNS, etc.* es opcional, siendo únicamente obligatorio, en lo relativo a puertos que se informe de si pertenece a *TCP, UDP o ICMP*.

Bro trabaja sobre Linux y sobre la terminal por lo tanto será necesario leer la documentación para saber que opciones nos interesa tener activadas cuando vayamos a analizar el tráfico, pues al no tener interfaz gráfica nos puede ocurrir que no tengamos activado lo que queramos y otras cosas que no precisamos si estén activas.

También es digno de mención que la información que hay en la red sobre Bro es bastante escasa, quedando casi exclusivamente la información que se puede encontrar en la web de Bro [3], por lo tanto a la hora de alguna duda, la

web será nuestra principal opción y en caso de no encontrar lo que queramos tendremos que hacer bastantes pruebas hasta dar con la clave. También podemos realizar una búsqueda por *GitHub* para coger ideas de cómo podemos resolver nuestras dudas. Aquí por ejemplo se puede encontrar información sobre cómo operar tablas con vectores de la manera correcta [4].

2.2.1. Requisitos de Bro

Podemos descargar los binarios y el código fuente desde la web de Bro [5], pero en mi caso después de experimentar ciertos fallos con ésta instalación, recomiendo seguir la instalación de Bro de su web en el apartado de instalación [6], pues allí nos guía a una instalación que es más fácil de mantener actualizada, ya que descargamos el código fuente directamente del **repositorio de GitHub**, por lo cual si hay algún cambio importante, con alguna mejora interesante la podremos tener casi directamente a nuestra disposición.

Para la instalación con los archivos de *GitHub*, tendremos que hacer lo siguiente:

```
sudo apt-get install cmake make gcc g++ flex bison libpcap-dev libssl-
    dev python-dev swig zlib1g-dev
```

Una vez que terminemos de instalar todo lo del comando anterior tendremos todos los prerrequisitos para la instalación de Bro, ahora clonamos el *repositorio de GitHub* en nuestro ordenador:

```
git clone --recursive git://git.bro.org/bro
```

Cuando termine de descargarlo nos metemos en la carpeta y ejecutamos lo siguiente:

```
./configure
make
make install
```

De esta forma ya tendremos instalado Bro en nuestro ordenador y podremos empezar a usarlo.

Ahora para usarlo tendremos que hacer lo siguiente, lo primero será **ajustar**

el PATH de nuestro terminal, para ello, tendremos que ejecutar en nuestro terminal lo siguiente:

```
export PATH=/usr/local/bro/bin:$PATH
```

Es importante hacerlo, pues sino obtendremos un error diciendo que Bro no se encuentra instalado. Una vez hecho esto, podemos empezar realizando un script sencillo y probar a ejecutarlo, un ejemplo de script sencillo es el que se ve en el apartado *Ejemplo de uso de evento*, supongamos que el script se llama *prueba.bro*, para lanzarlo tendremos que hacer lo siguiente:

```
~$ bro -b -r pcap/nitroba.pcap scripts/bro-flows/prueba.bro
```

En este caso dentro de la carpeta Bro que se genera cuando clonamos el repositorio he creado una carpeta llamada scripts, para tener todos los scripts allí guardados, por lo tanto al encontrarnos en la carpeta raíz de Bro, tendremos que ponerlo en la ruta, la opción *-b* que ponemos en la orden es solo para este ejemplo, lo que hace es no cargar los scripts que se encuentran en el directorio *base/*. Como necesitamos leer de un archivo *pcap*, tenemos que poner la opción *-r*, que indica a Bro que tiene un archivo de datos que leer, al igual que con los scripts, también hemos creado una carpeta pcap en la cual guardamos los archivos *pcap* que queramos leer. Una vez que lo ejecutemos se nos mostrará en el terminal un mensaje cada vez que se genere un flujo nuevo, cuando se exceda el timeout también y cuando un flujo vaya a ser eliminado de la memoria.

2.2.2. Gestión de tráfico en Bro

Nacimiento y muerte de flujos

En Bro el **nacimiento de un flujo** es controlado por un evento, dicho de otro modo, cuando aparece un nuevo flujo se lanza un evento, de ésta forma podremos controlar cuando un flujo nuevo aparece y podremos analizar toda su información en este evento o en los siguientes antes de que el flujo muera, o en el caso de tenerlo almacenado, antes de eliminarlo, pues una vez que lo eliminemos no podremos recuperarlo, por lo tanto para el análisis detallado del tráfico es interesante guardarlo.

La muerte de un flujo funciona también por eventos, siendo activado el evento cuando el flujo está a punto de morir, por lo tanto todavía podremos operar sobre él, por ejemplo, podremos decidir si queremos que el

flujo sea eliminado o seguir guardándolo para análisis estadísticos, realizar una comparación para ver si tenemos más flujos con sus características para reemplazarlo, etc.

Por lo tanto como se comentó en apartados anteriores en Bro la mayor parte de la información se gestiona con eventos, lo cual facilita enormemente el trabajo que tenemos por delante.

Marcas temporales

Es interesante remarcar el uso de los **timestamps**, pues como ya se sabe son una variable importante que usaremos en la comparación, esta variable es necesaria para emparejar los flujos, cierto es que los datos más importantes son las IP's y los puertos, pues si no son iguales no procederemos a aplicar la fórmula necesaria para el emparejamiento, pero llegado el momento también podríamos comparar los *timestamps* para comprobar si son distintos, pues en caso de que sean iguales podríamos desechar la comparación, pues básicamente sería comparar el mismo flujo, por algún error de programación y esto nos haría el programa más eficiente.

En Bro, como lenguaje de programación, volvemos a encontrarnos con un tipo de dato propio para la gestión de estos datos, en este caso el tipo es time [7], cuyo nombre es de por si muy indicativo de lo que va a contener. La peculiaridad de este tipo de dato es que el tiempo viene dado en valor absoluto, por lo cual tendremos que formatearlo para representarlo, aunque no será necesario si solamente queremos operar con él. Bro dispone de funciones para pasar de *double* a *time*, **double_to_time**, obtener el tiempo actual, **current_time**, y por último también disponemos de una función para obtener el tiempo del último paquete procesado, **network time**, obteniendo así su *timestamp*, aunque parezca que necesita estar procesando de manera online para usar esta función no es así, pues también sirve para los archivos de datos de tráfico de red guardados, como los archivos *pcap* que usaremos nosotros para el emparejamiento. Para mostrar correctamente el valor del tiempo podemos usar **strftime**, lo cual nos formateará el tiempo y lo mostrará acorde a lo que le indiquemos en la cabecera de la función. También contamos con el caso contrario, es decir, pasar de *string* a *time*, para ello tendremos que usar la función **strptime**.

Para acabar tenemos una de las funciones más importantes que es **time_to_double**, que sirve para pasar un tipo *time* a *double*, haciendo más fácil operar con él.

Listado de flujos activos

Para mantener un **listado de flujos activos** haremos uso de un contenedor para almacenarlos y haremos uso de los eventos de Bro para gestionarlos. Primero, cuando se generen flujos que previamente no habíamos visto los guardaremos para compararlos, si aparecen flujos que ya tenemos reconocidos, los compararemos con los almacenados y si nos interesa los guardaremos, y por último cuando Bro nos indique que algún flujo va a ser eliminado de la memoria y por lo tanto ese flujo ya no estará activo una vez que el evento termine, por lo tanto tendremos que operar con el flujo antes de que el evento termine, para ello veremos si hay algún candidato a ocupar su puesto en el listado de flujos emparejados y lo eliminamos del listado de flujos comparables.

Para mantener un listado de flujos activos Bro nos proporciona varias formas. La primera es usar un **vector**, el cual es fácil de usar si venimos de lenguajes como *C++*, podremos incluir datos usando los corchetes, *[ ]*, indicando el índice dentro de ellos, para borrar un elemento usaremos *delete* y para consultar el tamaño del vector usaremos *—v—*, pero sin embargo por si sólo es muy **ineficiente** para almacenar datos del tipo connection, pues solo dispone de índices numéricos.

Otro tipo para almacenar datos es **set**, el cual para borrar y consultar el tamaño es igual que para los vectores, sin embargo, para agregar datos tendremos que usar *add se[s]*, estando entre los corchetes lo que queremos almacenar, aunque también podemos almacenar datos redifiniendo el set, por ejemplo, *s1 = set(1,2,3)*. El set también es **ineficiente**, aunque menos, pues en un índice, aunque podemos guardar varios elementos estos tienen que ser únicos y corremos el riesgo de eliminar los datos almacenados al insertar nuevos datos.

Por último podemos usar un tipo *table*, que es básicamente un *map*, de los que se usan en *C++*, pudiendo hacerlos bastante interesantes, pues podemos hacer un **table de connection de vector** o de set de connection, por lo tanto esta será la forma que usaremos para almacenar el listado de flujos activos, pues nos da bastante versatilidad a la hora de manejar el listado de los flujos, porque podemos acceder por índice a la lista de todos los flujos que comparten datos, haciendo que sea una tarea bastante asequible, en términos de eficiencia.

2.2.3. Identificación de flujos en Bro (Tipo de flujo/Protocolo)

Como ya se explicó en otro apartado, Bro consta del tipo *connection* [8] para referirse y tratar los flujos, por lo tanto cómo en el flujo hay información sobre el protocolo que usa, podemos consultarlo haciendo referencia al flujo que estamos tratando. Hay que destacar que dentro de *connection*, en *history*, existe otro tipo llamado *Conn::Info* [9] que extiende aún más los datos de *connection*, y siendo capaz sus datos de crear los *logs* de lo que estemos analizando, por lo que podremos generar un log con los protocolos que son usados en la sesión. Cabe destacar que mostrando la información del puerto mediante el tipo *port*, el *string* que nos muestra lleva el número del puerto y el protocolo al que pertenece, pudiendo así extraer si es *TCP o UDP*. Aunque si lo que queremos es obtener solamente el protocolo de transporte al que pertenece el flujo podemos hacer uso de la función *get_port_transport_proto* [10].

Es destacable que el tipo de dato *port* soporta la comparación, lo cual será usado en el programa.

También en las múltiples extensiones de Bro encontramos un analizador de protocolos, protocol analyzer [**?**], el cual dispone de varias funciones interesantes para el análisis de protocolos, pero que nosotros no usaremos.

Después de lo expuesto en esta sección podemos decir que a partir del protocolo al que nos indique el flujo que pertenece, podremos clasificar mejor el tipo de flujo que estamos tratando.

2.2.4. Ejemplo de uso de eventos

Ahora que tenemos algunos conceptos claros pondremos el ejemplo de un programa en el cual se lanza un evento cada vez que aparece un flujo nuevo.

```
1 ## Para todo tipo de conexiones, ya sean TCP, UDP o ICMP. Este evento
      salta con cada nueva
2 ## conexion, con el primer paquete de una conexion desconocida, por lo
       que mirando el
3 ## documento generado vemos que esto es un nuevo flujo.
4 event new_connection(c: connection)
5   {
6
7    if ( connection_exists( c$id ) ) {
8       print fmt("Nueva conexion establecida Timestamp: %s desde %s a %
            s", strftime("%Y/%M/%d %H:%m:%S", network_time()),
            c$id$orig_h, c$id$resp_h);
```

```
9         print fmt("Protocolo del puerto: %s", get_port_transport_proto(
              c$id$orig_p));
10        print fmt("Informacion de las 4 tuplas del paquete: %s", c$id);
11      } else {
12        print fmt("La conexion ya existe");
13        print fmt("
              ------------------------------------------------------------
              ");
14      }
15
16   }
17
18 ## Solo disponible para conexiones TCP, se genera cuando no hay
       actividad en un
19 ## periodo de tiempo determinado.
20 event connection_timeout(c: connection)
21   {
22     print fmt("Conexion TCP ha excedido el timeout: %s",c$id);
23   }
24 ## Este evento salta para todo tipo de conexion, se da cuando el
       estado interno
25 ## esta a punto de eliminarse de memoria
26 event connection_state_remove(c: connection)
27 {
28
29   print fmt("Conexion de %s a %s a punto de ser eliminada de la
         memoria", c$id$orig_h, c$id$resp_h);
30
31 }
```

Algunos comentarios sobre este código. Como habrá podido notar en el lenguaje *Bro* los comentarios se realizan con dos almohadillas, *##*, los *print* son como en la gran mayoría de lenguajes, con la particularidad de que si vamos a mostrar un *string* debemos de indicar que va formateado mediante *fmt*, *%s* es para mostrar el *string*, tal y como se hace en *C* con los datos de tipo *int* mostrándolo con *%i*, por ejemplo. Por último para acceder a la distinta información que contiene el tipo de dato debemos de usar *$*, esto es parecido a como accedemos en *Java* a los datos de las clases.

Aunque el código está comentado merece la pena hacer algunas valoraciones sobre los eventos usados, el evento **new_connection** es muy útil, pues se ejecuta cada vez que llega un flujo que no ha sido usado antes y además reconoce *TCP, UDP e ICMP*. El evento **connection_timeout** aunque sólo sirva para flujos del protocolo *TCP* es bastante útil para empezar a trabajar sobre Bro, pues sirve para empezar a comprender como funcionan los eventos. Por último el evento **connection_state_removed** se lanza para *TCP, UDP e ICMP* cuando los flujos están a punto de morir, por lo tanto tiene un gran valor para eliminar de memoria los flujos que ya no nos interesan porque no están activos, y si este evento no se lanza no serán eliminados, en caso de que los flujos estén guardados.

Capítulo 3

Objetivos

En este trabajo lo que se tratará de realizar es:

- Demostrar que es posible analizar el tráfico mediante el emparejamiento de flujos.
- Realizar un programa que sea capaz, a partir de un archivo *pcap*, de analizar los flujos y emparejarlos.
- Que dicho programa muestre la información de los flujos emparejados.
- Implementación de la función de la comparación que se halla en el ensayo [1].

Para demostrar que es posible analizar el tráfico mediante el emparejamiento de flujos, y por tanto con DPI, se usará Bro para gestionar el tráfico de la red, creando un *script* para este cometido. Aunque se pretende realizar para una aplicación offline, también se podría usar para una aplicación en tiempo real sin problema.

Al realizar una **extensión** para Bro tendremos que adaptarnos y aprender a programar en el lenguaje que utilizan para ello el cual ya se nombré e incluso se ha visto algún ejemplo, Bro, el cual tiene ciertas peculiaridades, aunque como ya sabemos programar en distintos lenguajes, el aprendizaje de este lenguaje es muy rápido, pues la gran mayoría de las cosas ya sabemos realizarlas, sólo falta adaptarse a las peculiaridades que tiene, como ocurre con cualquier lenguaje. Alguna de estas peculiaridades es por ejemplo el acceso a los datos dentro de un tipo, que se realiza mediante *$*, cuando en otros lenguajes accedemos a ellos mediante *.* ó *-¿*.

Para mostrar la información de los flujos será necesario crear una función que nos muestre por pantalla que contienen dichos flujos. Realizar la función es para no repetir código, lo cual entra dentro de las buenas prácticas de programación. En dicha función se accederá a los datos de IP origen y destino

y los puertos origen y destino de los flujos que estamos comparando, esta función no realiza comparaciones, simplemente muestra la información, pues esta función es llamada después de encontrar dos flujos que son emparejables.

Para ver si dos flujos son emparejables, primero tendremos que ver si las IP's y los puertos de origen y destino son iguales. En caso de serlo pasaremos a aplicar la función, que se encuentra en el apartado *Estado del Arte*, la cual nos devolverá un número que compararemos con un umbral que nosotros definimos antes de empezar la ejecución, haciendo que si es mayor no sean emparejables y si es menor que dicho umbral si sean emparejables.

Capítulo 4

Implementación

Ahora vamos a proceder a explicar cómo se ha solucionado los problemas propuestos, siendo estos explicados a través del programa que se ha realizado, quedando así más clara la explicación.

Lo primero será mostrar que estructura se ha utilizado para almacenar los flujos.

```
1
2 ## Tablas para guardar los flujos que son emparejados
3 global collection: table[addr, addr, port, port] of vector of
      connection &synchronized;
4 global collection_added: table[addr, addr, port, port] of vector of
      connection;
5
6 ## El umbral: "Comparar la constante 'k', que es el umbral que fijare
      con el resultado que devuelve la funci n,
7 ## si es mas grande el resultado que 'k' se puede decir que los dos
      flujos son iguales, si es mas pequenio podemos decir que los dos
      flujos no son iguales"
8 ## resultado del umbral que calculamos
9 global umbral: double;
10
11 ## Definimos el umbral, de manera global para hacer las comparaciones
12 global k=10;
```

En este primer código tenemos que:

- Vamos a guardar los flujos en una tabla *collection* que tiene como índices dos direcciones IP y dos puertos, y a partir de estos indices tendremos un *vector de connection*.
- La tabla *collection_added* está destinada para guardar los flujos que son emparejados.
- El *umbral*, que será una variable global, y será donde almacenemos el

resultado de la función de comparación.

- k es el umbral que definimos nosotros al principio, que será el que usemos como referencia para saber si dos flujos son emparejables o no.

```
## funcion para la comparacion de los flujos, c1 el flujo que esta el primero en el vector de la tabla y c2 para el flujo que es candidato a ser emparejado

function emparejamiento(c1: connection, c2: connection ):double {

  ## A adimos variables para comprobar en la tabla, sin hacer bucle
  local orig = c1$id$orig_h;
  local dest = c1$id$resp_h;
  local po = c1$id$orig_p;
  local pd = c1$id$resp_p;

  local Nip = |collection[orig,dest,po,pd]|; ## Variable para saber cuantas conexiones tenemos
  local Po1: count; ## Puerto origen del primer flujo
  local Po2: count; ## Puerto origen del segundo flujo
  local Pd1: count; ## Puerto destino del primer flujo
  local Pd2: count; ## Puerto destino del segundo flujo
  local k1 = 1;  ## Variable fija
  local k2 = 10; ## Variable fija
  local dt: double; ## Variable para la diferencia de los tiempos
  local resultado = 0.0; ## Lo iniciamos a 0

  print c1$uid;
  print c2$uid;

  print fmt("Numero de Nip en table: %d", Nip);
  informacion_coincidencia(c1,c2);
  print fmt("Tiempo de inicio del flujo: %s", |c1$start_time|);
  print fmt("Tiempo de inicio del flujo: %s", |c2$start_time|);
  ## Para dp1 y dp2 que son 1-norm usamos la "Manhattan norm" que dice lo siguiente: SAD(x1,x2) = sumatoria(x1i - x2i)
  ## k1 y k2 son dos variables que nosotros le ponemos de forma manual, en este caso las pondremos como locales con 1 y 10 respectivamente
  ## dt es la diferencia de tiempo entre los time stamp de los primeros flujos de los flujos
  ## el tipo time se supone que es como un double, por lo tanto podremos restarlos sin problemas
  ## para la comparacion de puertos primero tendremos que hacer uso de la funcion  port_to_count [https://www.bro.org/sphinx/scripts/base/bif/bro.bif.bro.html#id-port_to_count]
  ## la cual nos pasa el puerto, que recordamos que va tambien con un string en el cual se nos dice que tipo es, a un
  ## valor numerico que si podremos restar sin problemas
  ## La funcion quedaria asi: (Nip-1)+(1/(dp1+k1))+(1/(dp2+k1))+(1/(dt+k2))
  Po1=port_to_count(c1$id$orig_p);
  Pd1=port_to_count(c1$id$resp_p);
  Po2=port_to_count(c2$id$orig_p);
  Pd2=port_to_count(c2$id$resp_p);
  ## local t1: double;
  ## local t2: double;
  ## t1 = time_to_double(c1$start_time);
  ## t2 = time_to_double(c2$start_time);
```

```
44
45   dt=(|c1$start_time| - |c2$start_time|);
46
47   ## print fmt("Tiempo paquete 1: %s", t1);
48   ## print fmt("Tiempo paquete 2: %s", t2);
49   print fmt("Diferencia de tiempo: %s", dt);
50
51   resultado=(Nip-1)+(1/((Po1-Po2)+k1))+(1/((Pd1-Pd2)+k1))+(1/(dt+k2));
52
53  return resultado;
54
55 }
```

En este trozo de código se muestra la función que se usa para devolver el resultado de la función de comparación. El resultado siempre es comparado con el umbral que hemos definido antes de lanzar el programa, y una vez que se usa esta función quiere decir que son *comparables*, pero no sabemos todavía si son *emparejables*. Si son emparejables lo sabremos solamente después de que hayamos ejecutado la función y se realice la comparación con el umbral. Para que quede más claro aquí esta el código de la comparación con el umbral.

```
1  . . .
2
3          umbral=emparejamiento(cl,c);
4      print fmt("connection_finished");
5      if(umbral>k){
6        ## Si el umbral calculado es mayor que el umbral de comparacion
             lo a adimos
7        print fmt("Si son emparejables TCP"); ## Mostramos TCP para
             saber en que evento se han calculado
8        collection[orig,dest,po,pd][|collection[orig,dest,po,pd]|] = c;
9        informacion_coincidencia(cl, c);
10       if( [orig,dest,po,pd] !in collection_added ){
11
12         collection_added[orig,dest,po,pd]=vector(c);
13         print fmt("A adimos una nueva conexion al vector de
               coincidencias");
14       } else {
15
16         ## Si ya esta, lo a adimos
17         collection_added[orig,dest,po,pd][|collection_added[orig,dest,
               po,pd]|] = c;
18         print fmt("Ya esta en vector de coincidencias y la a adimos")
               ;
19       }
20
21     } else{
22       ## Si el umbral calculado es menor que el umbral de comparacion
             no lo a adimos
23       print fmt("No son emparejables TCP");
24
25     }
26
27 . . .
```

En este ejemplo se calcularía para las conexiones de tipo *TCP*.

La comparación se dará cuando Bro reconozca un flujo de un determinado tipo de protocolo, en nuestro caso será mediante los eventos destinados a reconocer los tipos de protocolos de la *capa de transporte*. Dichos eventos son:

- connection_established, el cual se lanza cuando Bro localiza un SYN-ACK que corresponda a un handshake de TCP.
- connection_finished, el cual lanza Bro cuando una conexión TCP finaliza de forma normal.
- udp_request, el cual se lanza cuando Bro localiza un flujo que corresponde con UDP lanzado desde el origen.
- udp_reply, que es lanzado por Bro cuando localiza una respuesta a un flujo del anterior evento.
- icmp_echo_request, lanzado por Bro cuando localiza un flujo de tipo ICMP de echo.
- icmp_echo_reply, el cual es lanzado por Bro cuando localiza una respuesta a los flujos del evento anterior.

Dentro de estos eventos descritos se realiza la comprobación de que dos flujos se pueden comparar, ya que tienen los mismos IP's y puertos de origen y destino. Una vez en la función de comparación se almacenarán dichos puertos, pasándolos al tipo count para su uso en la función, se establecerán dos constantes, se calculará la diferencia de tiempo y en una variable se almacenará el número de flujos que hay con esos datos. Cuando realicemos esto, aplicaremos la función de cálculo y podremos decir si los flujos son emparejables.

Para mostrar la información de los flujos tenemos dos opciones, de las cuales mostramos el código a continuación:

```
1
2 ## Creo funcion auxiliar para ver la informacion del flujos que son
      coincidentes
3
4 function informacion_coincidencia(c: connection, p: connection){
5     print fmt("Informacion del primer flujo  IPo: %s , Po: %s , IPd: %
          s , Pd: %s ", c$id$orig_h, c$id$orig_p, c$id$resp_h,
          c$id$resp_p);
6     print fmt("Informacion del flujo coincidente  IPo: %s , Po: %s ,
          IPd: %s , Pd: %s ", p$id$orig_h, p$id$orig_p, p$id$resp_h,
          p$id$resp_p);
7 }
```

```
8
9  ## Funcion auxiliar para mostrar la informacion de un solo flujo
10
11 function informacion_flujo(c: connection){
12     print fmt("Informacion del flujo a adido IPo: %s , Po: %s , IPd:
           %s , Pd: %s, uid: %s ", c$id$orig_h, c$id$orig_p, c$id$resp_h,
            c$id$resp_p, c$uid);
13 }
```

La función informacion_flujo sirve para mostrar la información del flujo relativa a las IP's y a los puertos que contiene, así como el *UID* del flujo, que es el identificador único que se le asigna automáticamente a cada flujo.

Por su parte la función informacion_coincidencia sirve para mostrar la información de dos flujos a la vez, mostrando sus IP's y sus puertos. Esta función será usada para mostrar la información de los dos flujos que son emparejables, siendo el primero de ellos el que se usa para comparar siempre y el segundo es el nuevo flujo que se ha detectado. De esta forma tan sencilla se puede mostrar toda la información interesante sin tener que recorrer al final de la ejecución del programa todo la *tabla de connection*.
Para eliminar los flujos que, por su *timestamp* estén a punto de morir, Bro nos da la opción mediante un evento de gestionar que hacer antes de que sean eliminados de la memoria, dicho evento es:

- connection_state_removed, evento que se lanza cuando un flujo activo va a ser eliminado de la memoria.

Cabe destacar que el borrado no es todavía operativo, por lo que en el script puede que este evento esté comentado.

```
1  ## Generated when a connections internal state is about to be
       removed from memory. Bro generates this event reliably
2  ## once for every connection when it is about to delete the internal
       state. As such, the event is well-suited for
3  ## script-level cleanup that needs to be performed for every
       connection.
4  ## This event is generated not only for TCP sessions but also for UDP
       and ICMP flows.
5
6  event connection_state_remove(c: connection){
7
8    local orig = c$id$orig_h;
9    local dest = c$id$resp_h;
10   local po = c$id$orig_p;
11   local pd = c$id$resp_p;
12   local coleccion = collection[orig,dest,po,pd];
13   local tama=|coleccion|;
14
15
16   for(j in coleccion){
```

```
17      if(j+1 >= tama){
18        if(tama==1){
19          collection[orig,dest,po,pd]=vector();
20        }
21        if(j+1==tama){
22          delete collection[orig,dest,po,pd][tama]; ## Aqui esta el
                error
23        }
24        break;
25      } else {
26        collection[orig,dest,po,pd][j]=coleccion[j+1];
27      }
28    }
29
30    print fmt("Terminamos copia y borrado...");
31
32  }
```

En este evento lo que tratamos de hacer es, antes de eliminarlo de la memoria, ver si existe algún flujo que pueda ocupar su lugar siendo el flujo referencia a la hora de comparar flujos. Básicamente lo que hace es comprobar si el tamaño es mayor que 1. Si es 1 solamente tendremos que borrar el vector contenido en la posición de la tabla, y si tenemos que es mayor, tendremos que borrar el primero y mover todos una posición hacia atrás. Nos interesa que este evento funcione, pues si los *timestamps* de dos flujos tienen mucha diferencia los daremos como que no son comparables, cuando si se van actualizando, tendremos más posibilidades de que la clasificación sea correcta.

Para añadir flujos a la tabla de conexiones, tenemos un evento dedicado a ello:

- new_connection, Bro lanza este evento cada vez que detecta un flujo que era desconocido.

```
1  ## Cada vez que entra un nuevo flujo compruebo que si esta en la tabla
2  ## Este evento se lanza con cada nueva conexion de un flujo que no sea
       conocido, pero al borrarlo de memoria ser  desconocido aunque ya
       lo tengamos en la tabla
3  ## Generated for every new connection. This event is raised with the
       first packet of a previously unknown connection. Bro uses a flow-
       based definition of  connection  here that includes not only
       TCP sessions but also UDP and ICMP flows.
4
5  event new_connection(c: connection){
6
7    local orig = c$id$orig_h;
8    local dest = c$id$resp_h;
9    local po = c$id$orig_p;
10   local pd = c$id$resp_p;
11   print fmt("new_connection");
```

```
12
13   if( [orig,dest,po,pd] !in collection ){
14
15     ## Si no estan los valores clave del flujo lo creamos
16     collection[orig,dest,po,pd]=vector(c);
17     informacion_flujo(c);
18     print fmt("A adimos una nueva conexion");
19
20   } else {
21
22     ## Si ya esta, lo a adimos
23     collection[orig,dest,po,pd][|collection[orig,dest,po,pd]|] = c;
24     informacion_flujo(c);
25     print fmt("Ya esta y la a adimos");
26
27   }
28
29 }
```

Lo que se realiza en este evento es ver si en la tabla global tenemos ya almacenado un flujo de esas características. Si no tenemos uno igual creamos en la tabla para los valores de las IP's origen, destino y puertos de origen y destino un nuevo vector que contenga solamente este nuevo flujo. En caso de que si tengamos los valores ya almacenados solamente tendremos que añadir al final del vector el nuevo flujo. En ambos casos mostraremos la información del flujo.

Por último usamos dos eventos en el programa para ver cuando se lanza y cuando finaliza el programa:

- bro_init, evento que se lanza cuando iniciamos Bro.
- bro_done, evento que se lanza cuando Bro va a terminar.

```
1  ## Evento que se lanza cuando se inicia BRO
2
3  event bro_init(){
4
5    print fmt("Hora de inicio: %s", current_time());
6
7  }
8
9  ## Evento que se genera cuando BRO va a tenerminar
10
11 event bro_done(){
12
13
14   print fmt("Hora de finalizacion: %s", current_time());
15
16 }
```

En el primer evento, que es el que se lanza cuando Bro se ejecuta, mostramos la hora de inicio. En el segundo evento, que solamente se ejecuta cuando Bro va a finalizar, mostramos la hora de finalización.

Capítulo 5

Conclusiones y trabajo futuro

5.1. Conclusiones

Después de lo visto anteriormente se puede llegar a la conclusión de que mediante un NMS, al cual mediante la agregación de funcionalidades, se le proporciona la capacidad de clasificar el tráfico de una red mediante emparejamiento de flujos. En nuestro caso los protocolos analizados son solamente **TCP y UDP**, es decir, los protocolos de la capa de transporte y en algunos casos también el protocolo ICMP.

Fue muy sencillo poder mostrar los datos de los flujos, pues que se disponga de un tipo de dato que almacene todos los datos correspondientes al flujo hace que trabajar con ello sea muy fácil, incluso mostrar el de todos los flujos que pasan por el programa, y que el rendimiento no se vea afectado. Puede considerarse un inconveniente que no disponga de interfaz, pues en el terminal se puede llegar a confundir las cosas dependiendo de la velocidad con la que saque la información, que sea un lenguaje basado en eventos también puede llegar a ser un problema a la hora de mostrar la información, pues puede ser que estemos mostrando la información de dos flujos que estamos comparando de tipo UDP por ejemplo, y que salte un evento de TCP y se muestre mientras todavía se está mostrando la información del evento de UDP.

5.2. Trabajo futuro

Obviamente este programa al solo poder emparejar flujos mediante el protocolo de transporte se puede quedar corto a la hora de querer obtener un análisis más exhaustivo, de modo que inspeccionemos los paquetes de una forma más profunda, de modo que a este programa se le puede agregar

más funcionalidades haciendo que sea más completo, y clasifiquen además los flujos por el tipo de protocolo en la capa de aplicación pertenecen, con esto nos referimos a *HTTP, SMTP, FTP, DNS y demás*. Esto podría llegar a realizarse analizando más detenidamente los puertos. Esta extensión se podría realizar con algunos clasificadores de puertos que ya están realizados y se pueden encontrar con cierta facilidad en *GitHub*.

Bibliografía

[1] José Camacho, Pablo Padilla, Pedro García-Teodoro, and Jesús Díaz-Verdejo. A generalizable dynamic flow pairing method for traffic classification.

[2] Dr. Thomas Porter. The perils of deep packet inspection. URL https://www.symantec.com/connect/articles/perils-deep-packet-inspection.

[3] Bro Team. Bro indice, . URL https://www.bro.org.

[4] securitykitten. Finding beacons with bro. URL https://gist.github.com/securitykitten/a7edcee0932c556d5e26.

[5] Bro Team. Descarga de bro, . URL https://www.bro.org/download/index.html.

[6] Bro Team. Instalación de bro, . URL https://www.bro.org/sphinx/install/index.html.

[7] Bro Team. Tipo time, . URL https://www.bro.org/sphinx/script-reference/types.html?highlight=time#type-time.

[8] Bro Team. Tipo cennection, . URL https://www.bro.org/sphinx/scripts/base/init-bare.bro.html#type-connection.

[9] Bro Team. Tipo conn, . URL https://www.bro.org/sphinx/scripts/base/protocols/conn/main.bro.html#type-Conn::Info.

[10] Bro Team. Función get_port_transport_proto, . URL https://www.bro.org/sphinx/scripts/base/bif/bro.bif.bro.html#id-get_port_transport_proto.

[11] Bro Team. Web de bro, . URL https://www.bro.org/sphinx/intro/index.html.

[12] Bro Team. Analizadores de protocolos, . URL https://www.bro.org/sphinx/script-reference/proto-analyzers.html.

[13] James F. Kurose and Keith W. Rose. *Redes de computadores un enfoque descendente*. Pearson, 2010.