4. Snort

4.1. Introducción

En este capítulo se explicará en detalle todo lo relativo a Snort. Para entender la aplicación en profundidad no se pueden obviar sus orígenes ni la filosofía que ha perseguido desde sus inicios Martin Roesch, su creador.

No menos importante será comprender la situación de la época, las necesidades que existían y qué tipo software podía plantearse como una solución interesante que supliera ciertas carencias. Se hablará también del concepto de Open Source, que ha ido de la mano de Snort desde sus inicios y al que hoy en día aún permanece fiel.

Una vez repasada la historia de Snort, sus orígenes y la filosofía que ha seguido, en lo que resta de capítulo se hará un recorrido por las entrañas de Snort, estudiando sus componentes principales y conociendo el rol que juega cada uno. En primer lugar, realizando una descripción teórica para cada componente y posteriormente situando y analizando cada uno de ellos en el código fuente de Snort. Esta explicación, a nivel de desarrollador, facilitará posteriormente la comprensión del trabajo realizado durante el proyecto.

4.2. Historia

En una arquitectura de seguridad de red los Sistemas de Detección de Intrusos de Red (NIDS) son una parte importante que monitoriza el tráfico de la red, detectando actividades sospechosas y alertando a los administradores del sistema de la presencia de posible tráfico hostil. Los primeros NIDS comerciales cumplían con esta labor pero contaban con diferencias importantes entre ellos, lo que resultaba una solución muy cara para cualquier organización que quisiera contar con un sistema que aprovechara los aspectos más destacados de cada uno. Snort fue diseñado para tratar de solventar estos problemas.

Las ideas originales en las que se basa el desarrollo de Snort vienen del movimiento Open Source, cuyo pionero fue Richard Stallman durante la década de los 80. Open Source defiende la idea de que todo software debería proporcionar su código fuente y a su vez ser desarrollado por comunidades formadas por personas interesadas en el proyecto. Esta ideología y la convicción de que el camino del software libre proporciona mejores aplicaciones fue elaborada en profundidad en la obra de Eric S. Raymond, “The Cathedral and the Bazaar”, considerado como el tratado fundamental del desarrollo de software libre. Raymond explica cómo una aplicación desarrollada en

base a ideas Open Source puede ser superior a una equivalente que siga métodos propietarios tradicionales. El proyecto Snort confía en esta ideología para llevar a cabo su desarrollo y hasta ahora ha conseguido estar a la cabeza, o en una posición muy alta, de las tecnologías basadas en detección y prevención de intrusos.

Snort fue lanzado en 1998 por Martin Roesch e inicialmente declarado como un "lightweight" intrusion detection technology. Un lightweight IDS debería ser multi-plataforma, tener un bajo impacto en el sistema y ser fácilmente configurable por un administrador de sistemas que necesite implementar una solución de seguridad específica en un breve intervalo de tiempo. Por aquel entonces, Snort llenó un importante hueco en el ámbito de los sistemas de seguridad de red. Se trataba de una aplicación ligera capaz de monitorizar pequeñas redes TCP/IP y de detectar una amplia variedad de tráfico sospechoso, así como ataques ya conocidos. También podía proporcionar suficiente información a los administradores del sistema para que tomaran decisiones en cuanto a actividades sospechosas. Snort también tenía la capacidad de solventar rápidamente los agujeros potenciales que pudieran surgir en un entorno de seguridad de red, ya que cuando se descubría un ataque los sistemas propietarios tardaban bastante tiempo en publicar una nueva base de firmas que lo solventara, mientras que con Snort se podía actuar con mayor rapidez.

Básicamente, Snort es un analizador de paquetes de red que funciona como sistema de detección de intrusos y está basado en la librería libpcap, una interfaz de sistema para la captura de paquetes que se creó como parte de la aplicación tcpdump. Esta librería permite a los desarrolladores poder recibir paquetes de la capa de enlace de datos y trabajar sobre ellos. La principal característica que distingue a Snort de tcpdump es la inspección del payload del paquete. Esto permite a Snort detectar un gran abanico de actividades hostiles, incluyendo buffer overflows, escaneo de vulnerabilidades CGI o cualquier otra que pueda ser detectada en el payload.

En la actualidad, Snort ha evolucionado hasta ser considerado un estándar de facto en prevención y detección de intrusos, contando con más de 4 millones de descargas y cerca de 400.000 usuarios registrados y convirtiéndose en la tecnología de prevención de intrusos con mayor despliegue mundial. Usa un lenguaje de reglas flexible para detectar tráfico en tiempo real, así como un motor de detección basada en arquitectura modular.

El poder y expansión de Snort se debe en gran parte a la influencia y al alcance de la comunidad de usuarios de Snort, ya que entre ellos existe un gran número de programadores que testean y publican resultados y opiniones acerca de las funcionalidades de Snort y del conjunto de reglas. Como aventuró Eric Raymond en su obra, y posteriormente se comprobó en el desarrollo de GNU/Linux, cuando en una comunidad Open Source se detectan fallos, se responde ante ellos de forma más rápida y eficiente que en un entorno de desarrollo propietario.

Gracias al hecho de ser una aplicación Open Source, Snort cuenta con la ventaja de ser un sistema configurable y adaptable a necesidades concretas, por lo que puede ser una buena solución si se busca un sistema personalizado. Este es uno de los motivos por los que grandes organismos, como gobiernos y organizaciones militares, han decidido implementar sus propios sistemas de detección de intrusos utilizando Snort en lugar de aplicaciones propietarias que en muchos casos no alcanzan el mismo rendimiento ni las prestaciones de Snort.

4.3. Componentes de Snort

En el desarrollo de una nueva aplicación, una práctica común en el mundo Open Source es escoger una herramienta ya existente y evolucionar a partir de ella para crear algo nuevo. En el caso de Snort, esta herramienta se trataba de tcpdump, un capturador de paquetes muy extendido en sistemas Unix, el cual obtenía los paquetes directamente de la capa de Enlace de Datos, a través del uso de las librerías libpcap, como se muestra en la figura 4.1. Martin Roesch, el creador de Snort, aprovechó de tcpdump la habilidad de capturar paquetes y añadió la posibilidad de analizarlos en busca de ataques conocidos cuyas descripciones formaran parte de una base de datos de firmas. En eso consistía, muy resumidamente, su nueva aplicación, que con el paso de los años ha crecido en cuanto a complejidad y funcionalidades.

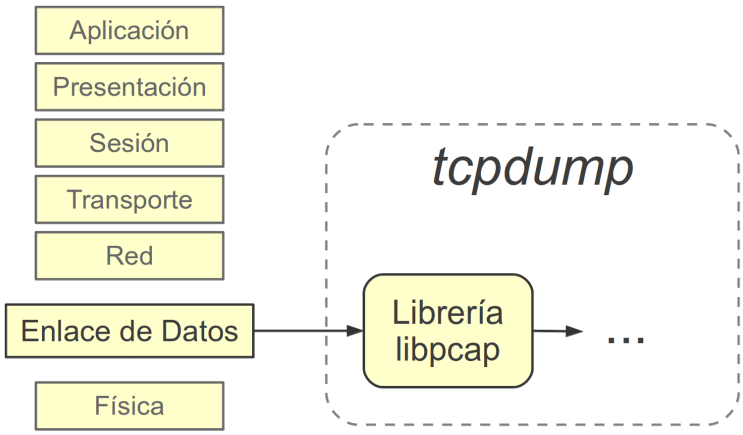

Figura 4.1: Modelo OSI y captura de paquetes por parte de tcpdump a través de las librerías libpcap.

A partir de la versión 2.9 de Snort se introduce la librería de adquisición de datos, más conocida como DAQ (del inglés Data Acquisition). El objetivo del módulo DAQ es remplazar las llamadas directas a las funciones de la librería libpcap por una capa abstracta, para así facilitar la ejecución de operaciones en diversos entornos hardware y software sin necesidad de realizar cambios en Snort. Por tanto, a partir de entonces el módulo DAQ ha sido el encargado de redirigir los paquetes a Snort, en lugar de hacerlo el propio Snort a través de las librerías libpcap.

Snort está formado por varios componentes que se muestran en la figura 4.2, cada uno de ellos es el encargado de llevar a cabo una tarea específica. En el momento en que circula un paquete por la red, este es capturado por el módulo DAQ, que lo reenvía posteriormente a Snort. A partir de aquí, recorre cada componente de Snort en el siguiente orden:

1. Packet Decode Engine (motor de descodificación de paquetes o simplemente descodificador de paquetes): Una vez que el modulo DAQ envía el paquete a Snort, este pasa por el descodificador de paquetes (Packet Decoder), que se encarga de almacenar toda la información de cada paquete que llegue de la red, como por ejemplo protocolos, IPs de origen y destino, etc., en una estructura de datos para su posterior procesamiento.

2. Preprocessor (preprocesador): Los preprocesadores fueron introducidos en la versión 1.5 de Snort. Gracias a ellos se pueden ampliar las funcionalidades de Snort permitiendo a usuarios y programadores crear módulos (plug-ins) dentro de Snort de una manera sencilla. Una vez que se descodifica el paquete, se ejecutan los preprocesadores, que pueden analizar e incluso modificar el paquete en cuestión, dependiendo del objetivo de cada preprocesador. También, pueden lanzar alertas, clasificar o descartar un paquete antes de enviarlo al motor de detección (Detection Engine), que cuenta con un alto coste computacional.

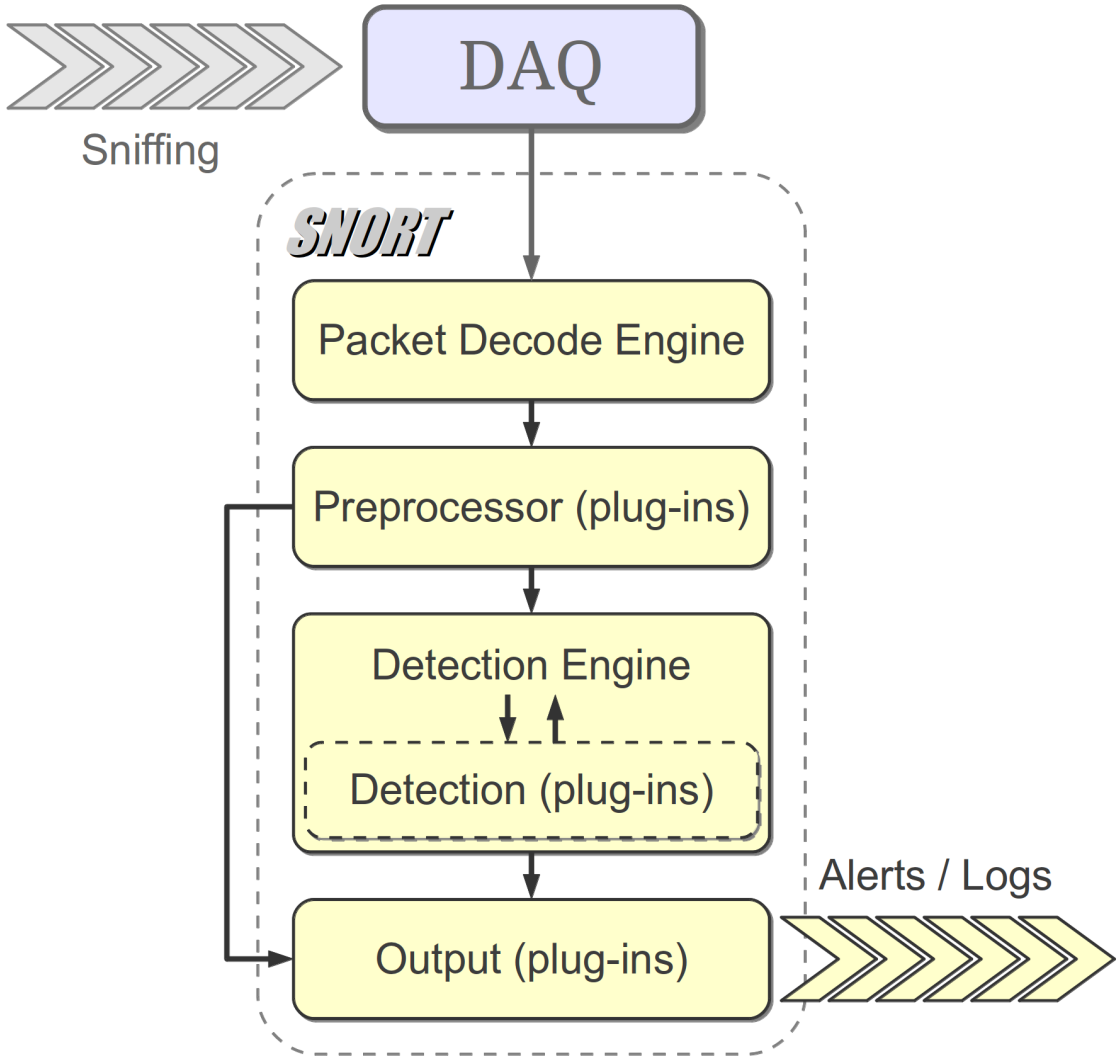

Figura 4.2: DAQ y Componentes de Snort. Flujo de datos desde la capa de enlace de datos hasta la salida de Snort.

3. Detection Engine (Motor de detección): Este componente es considerado como el corazón de Snort. Toma información del Packet Decoder y de los preprocesadores e inspecciona el contenido del paquete para compararlo a través de su módulo, o plug-in, de detección (Detection) con los patrones de la base de firmas.

4. Output (Salida de Snort): Cuando se ha detectado un paquete sospechoso, ya sea porque un preprocesador lo ha decidido o porque cumple con una regla concreta, este módulo de salida genera una alerta, en el formato que se especifique en el archivo de configuración de Snort.

A continuación se explicarán los componentes que son considerados más importantes: el descodificador de paquetes, el preprocesador y el motor de detección.

4.3.1. Descodificador de paquetes

Una vez que los paquetes son capturados de la red por el módulo DAQ y son reenviados a Snort, se necesita descodificar la información de cada paquete de la capa de enlace de datos para su posterior procesamiento. Snort tiene la capacidad de reconocer diferentes protocolos como Ethernet, 802.11, Token Ring, etc., así como protocolos de capas superiores como IP, TCP y UDP. Durante la fase de descodificación Snort organiza en estructuras de memoria los diferentes campos del paquete para su posterior análisis por los preprocesadores y el motor de detección. En la figura 4.3 se muestra un esquema del camino a seguir por un paquete en la fase de descodificación.

1. Cuando Snort arranca, la tarjeta de red se configura en modo promiscuo a través de la librería libpcap, que es invocada por el módulo DAQ.
2. El módulo DAQ funciona como un bucle infinito que se encarga de esperar que circulen paquetes por la red para capturarlos y reenviarlos a Snort.

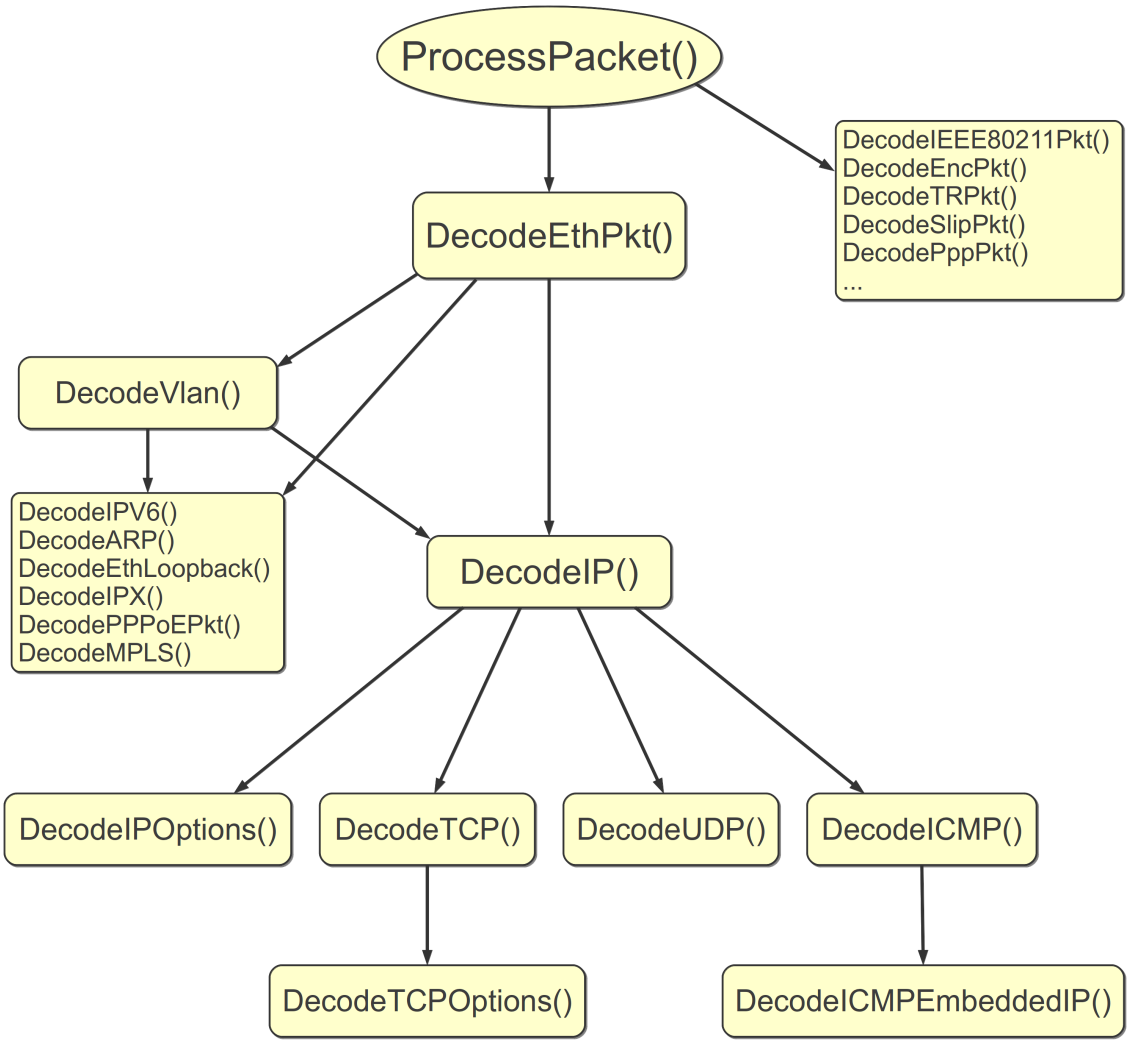

Figura 4.3: Esquema representativo del camino seguido por un paquete en su descodificación.

3. Cuando Snort recibe un paquete, llama a la función `ProcessPacket` que será la encargada de gestionar la descodificación del paquete.
4. Cuando se trate de una trama Ethernet, la función anterior llamará a `DecodeEthPkt`, que descodificará la trama.
5. Una vez dentro de la función `DecodeEthPkt`, ésta llamará a las funciones correspondientes. Dependiendo del tipo de trama Ethernet que se trate y la información que contenga el paquete se enlazará a las estructuras de datos apropiadas. Una vez hecho esto, ya se podrá llamar al siguiente componente de Snort.

4.3.2. Preprocesador

En este punto de la ejecución, cuando ya se han constituido las estructuras de datos que contienen la información de los paquetes, se podrían analizar y tratar los datos que circulan por la red en el motor de detección. No obstante, antes de que los paquetes vayan hacia el motor de detección, hay que verificar si realmente tales paquetes se deben analizar y qué tipo de análisis se debe llevar a cabo. De esta tarea se encarga el preprocesador de Snort. En la figura 4.4 se representa un esquema general del funcionamiento que sigue el preprocesador.

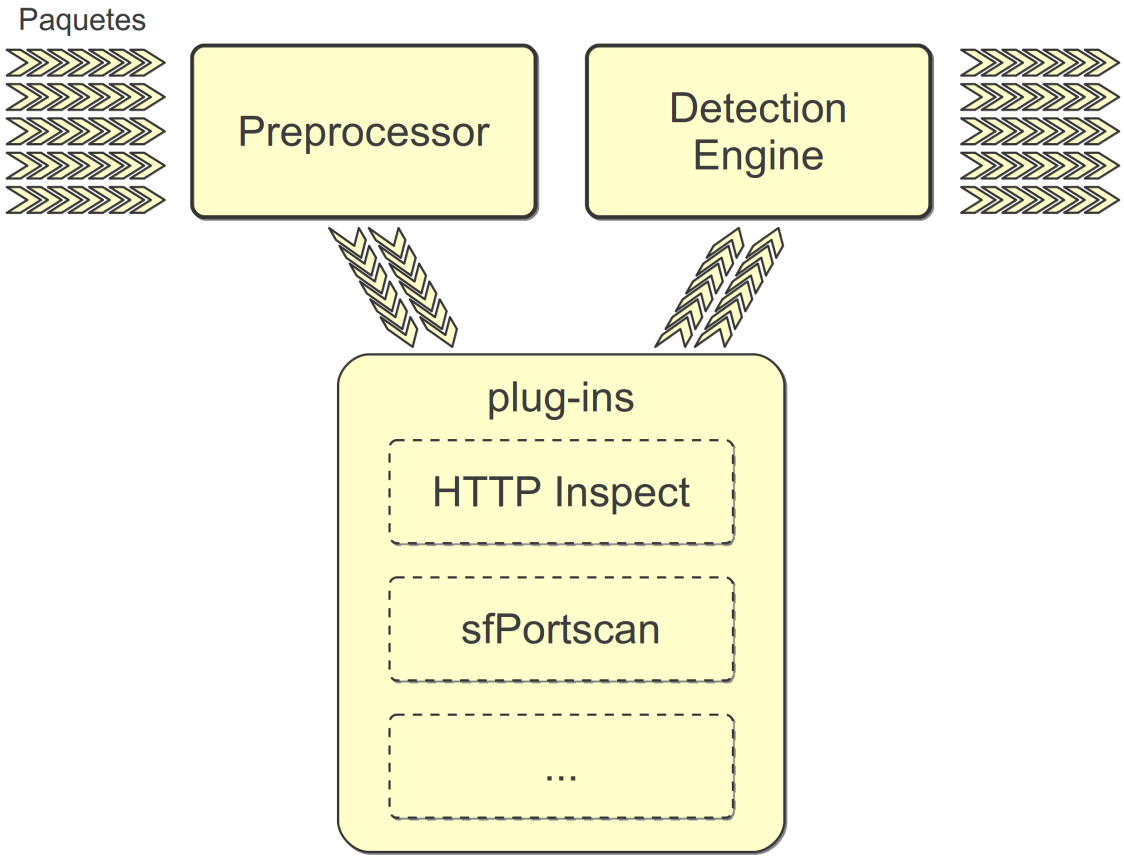

Figura 4.4: Funcionamiento del preprocesador de Snort.

Por tanto, antes de enviar cada paquete entrante al motor de detección, el preprocesador se encargará de analizar cada uno de ellos contra cada plug-in que haya sido activado previamente en el archivo de configuración de Snort. Estos plug-ins buscan un cierto tipo de comportamiento por parte del paquete y una vez que se determina, éste se envía al motor de detección. Por ejemplo, si por cualquier razón no se quiere analizar el tráfico RPC que circula en la red, se puede desactivar el plug-in correspondiente en el archivo de configuración de Snort y dejar el resto de plug-ins activos. El hecho de que el usuario tenga la posibilidad de poder activar y desactivar los módulos (plug-ins) que desee es una gran ventaja para un IDS.

El concepto de preprocesador nació en la versión 1.5 de Snort. La principal idea que perseguía la inclusión de preprocesadores fue la de incorporar la posibilidad de permitir inspeccionar los paquetes antes de que alcanzaran el motor de detección, para así generar alertas sobre los paquetes, omitirlos del análisis si cumplieran, o no, con ciertas características y también tener la posibilidad de modificar el contenido de los mismos antes de reenviarlos al motor de detección.

A continuación, se presentarán algunos de los preprocesadores existentes en Snort, como son el HTTP Inspect, Frag3 y Stream5, todos muy importantes para un funcionamiento eficiente de Snort. De esta forma se entenderá mejor la utilidad que aportan a la aplicación. Otros preprocesadores, como el sfPortscan, también forman una parte importante de Snort. A cualquier desarrollador que trabaje con Snort se le recomienda el estudio en profundidad de todos y cada uno de ellos; lo explicado en este documento tiene únicamente carácter introductorio.

4.3.2.1. Preprocesador HTTP Inspect

Un ejemplo de utilidad de este preprocesador se muestra en la figura 4.5. Supongamos una situación corriente de Cliente-Servidor y una base de firmas en Snort que contenga la regla `ataque.exe`. Si el cliente se dispusiera a realizar un ataque contra el servidor web e intentara ejecutar remotamente un fichero malicioso introduciendo en su navegador web la dirección `http://servidor/ataque.exe`, Snort detectaría el paquete y lanzaría una alerta del ataque producido. Suponer ahora que el cliente intenta saltar la protección que ofrece Snort y realiza una petición web de la dirección `http://servidor/%61%74%61%71%75%65%2E%65%78%65`. El protocolo HTTP permite el uso de caracteres binarios en la URI mediante el uso de la notación `%XX`, donde `XX` es el valor en formato hexadecimal del carácter codificado en ASCII. Cuando la URI llega al servidor web, éste la convierte en `http://servidor/ataque.exe`, que ejecutaría un código malicioso. En el caso de que en Snort no existiera un módulo de preprocesador encargado de descodificar este tipo de URIs el motor de detección buscaría el patrón `ataque.exe` sin tener éxito y por tanto marcaría el paquete como benigno, descartando de este modo el envío de alertas al administrador. El caso descrito se muestra en el modo de uso de Snort que se puede ver en la parte alta de la figura, donde no se usa el preprocesador HTTP Inspect. En el caso inferior, cuando sí se utiliza dicho preprocesador, que descodifica la URI, se envía al motor de detección el paquete con la URI descodificada, de esta forma se detecta el ataque y se informa debidamente al administrador del sistema.

Otros preprocesadores como el RPC Decode o el FTP/Telnet Preprocessor realizan similares tareas de normalización, sólo que en lugar de normalizar URIs, estos preprocesadores normalizan tráfico RPC, FTP y Telnet antes de enviarlo al motor de detección de Snort.

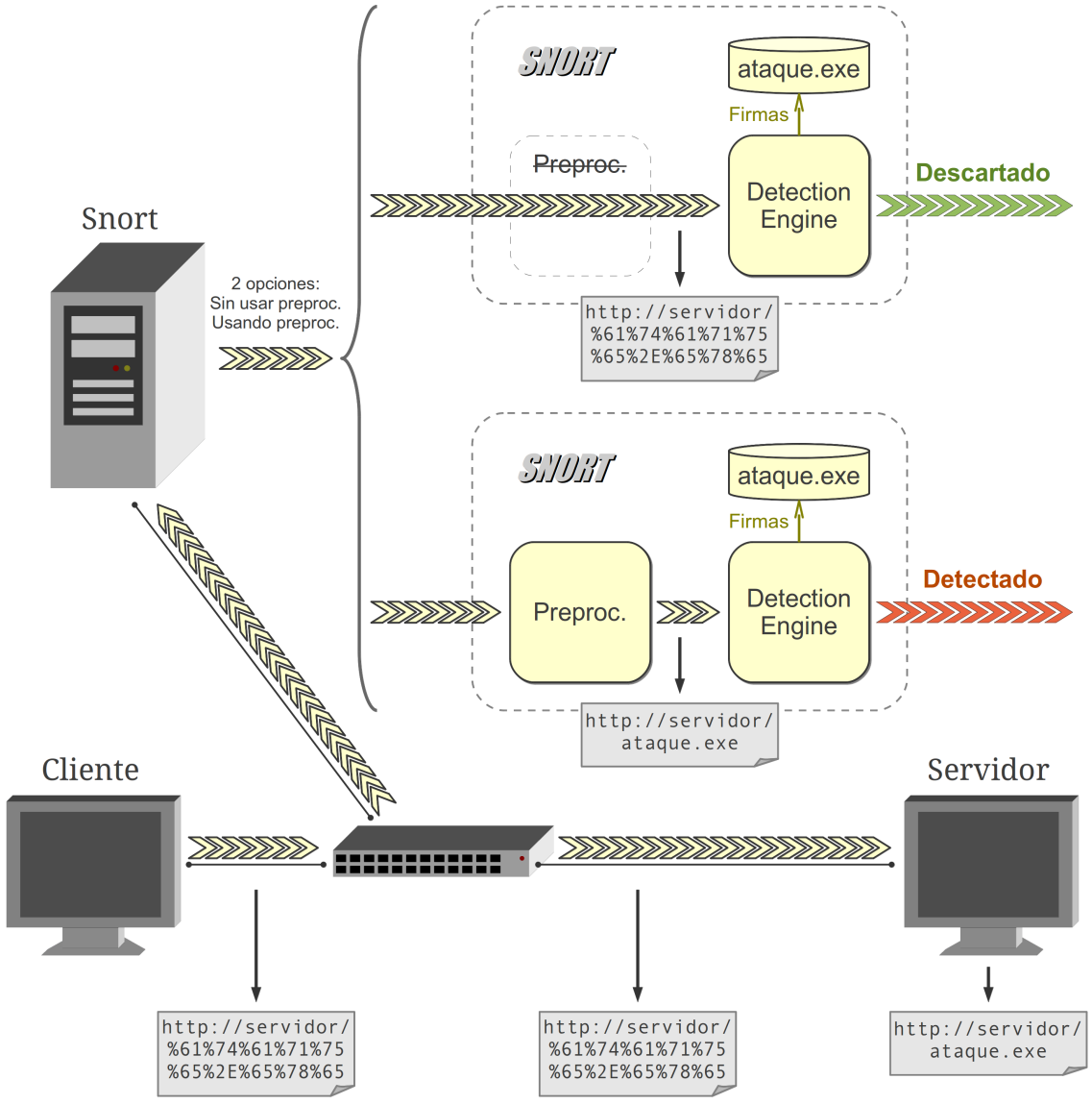

Figura 4.5: Ejemplo que demuestra la utilidad del componente preprocesador de Snort.

4.3.2.2. Preprocesador Frag3

La fragmentación de paquetes es un proceso necesario en toda red IP. Cuando un paquete IP supera en longitud a la unidad máxima de transferencia (MTU, Maximum Transfer Unit) éste debe ser fragmentado en tantos trozos como sea necesario para poder circular por la red. Este aspecto suele ser usado por atacantes para evitar ser detectados por IDSs.

Un posible uso malicioso de la fragmentación de paquetes puede ser el de realizar ataques de denegación de servicio (DoS: denial-of-service). En el caso de que un atacante quisiera "tumbar" un servidor web, podría enviarle paquetes fragmentados en trozos excesivamente pequeños para que de esta forma el servidor tuviera que dedicar una gran cantidad de recursos para reensamblarlos.

Una medida para evitar ataques de este tipo podría ser clasificar los paquetes fragmentados como sospechosos y bloquearlos directamente en los routers o firewalls de la red. Sin embargo, la existencia de paquetes fragmentados en una red es bastante normal, más aún si la red está conectada a Internet. Por lo tanto, actuando de esta forma se podría estar bloqueando el acceso a usuarios bienintencionados que estuvieran accediendo a la red a través de enlaces con MTUs bajos. En conclusión, habría que tratar de buscar una solución de compromiso entre el grado de seguridad y la funcionalidad de la red.

Otro tipo de ataque que aprovecha la fragmentación de paquetes, y por el cual cobra sentido el uso del preprocesador Frag3, es la inclusión de un ataque fragmentado y repartido en más de un paquete. Para su comprensión podrá servir de ayuda repasar la distribución del esquema de la figura 4.5 pero en la variante mostrada de la figura 4.6.

Llegados a este punto, el cliente habría descubierto que su ataque había sido detectado por Snort gracias al uso del preprocesador HTTP Inspect, que analizaba y convertía la URI para que pudiera ser analizada correctamente por el motor de detección. Por tanto, para atacar de nuevo al servidor web deberá elegir otro método, en este caso se decanta la fragmentación IP. El atacante decide fragmentar el paquete antes de enviarlo para que de esta forma la palabra `ataque.exe` se reparta entre dos paquetes y el motor de detección de Snort no la detecte y no emita una alerta. Al llegar al servidor web, este reensamblará los paquetes y ejecutará la petición que resultará ser un ataque. El preprocesador Frag3 se encarga de enviar al motor de detección los paquetes reensamblados para que de esta forma este tipo de ataques puedan ser detectados.

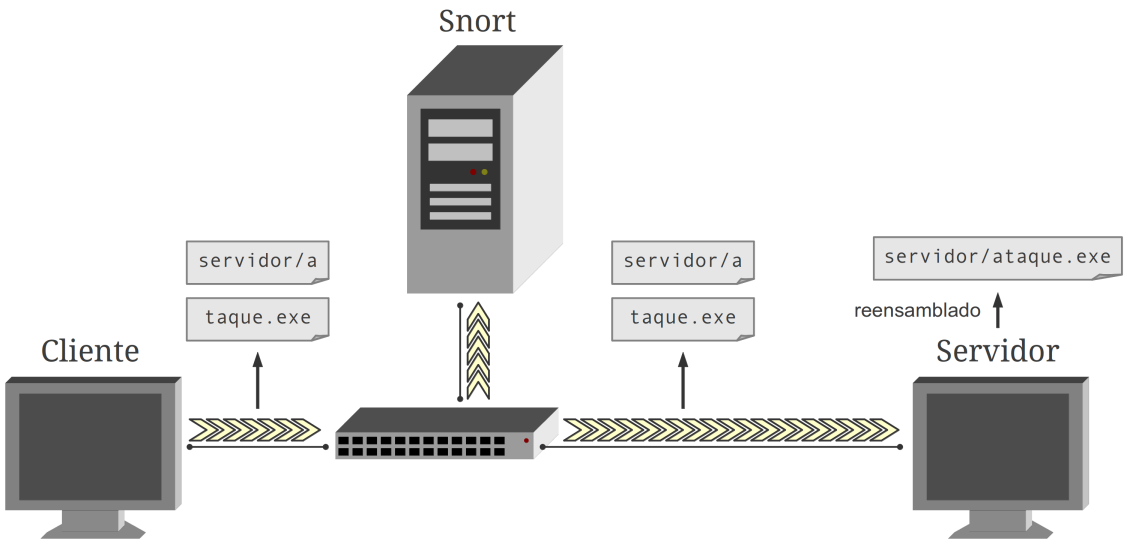

Figura 4.6: Esquema que muestra la utilidad del preprocesador Frag3.

4.3.2.3. Preprocesador Stream5

A veces, sólo interesa analizar paquetes que formen parte de una conexión establecida, para ello Snort debe seguir el estado de las sesiones en vigor. Este es el objetivo del preprocesador Stream5, que fue diseñado para hacer de Snort una aplicación orientada a conexión.

Una de las ventajas de conseguir que Snort esté orientado a conexión es que de esta forma se pueden detectar ciertos ataques que usen aplicaciones como *stick* con el objetivo de no ser detectados por Snort. Esta aplicación realiza un estudio de las firmas de Snort y se encarga de generar paquetes que provoquen el lanzamiento de alertas por parte del IDS. De esta forma se consigue sobrecargar a Snort con el fin de que se vea obligado a omitir muchos paquetes del análisis en el motor de detección; es en este momento donde se lanzaría el verdadero ataque.

El modo en el que el preprocesador Stream5 consigue evitar este tipo de ataques consiste en mantener un control de las conexiones establecidas en tiempo real. Para ello, el preprocesador construye tablas internas que representan dichas sesiones y las elimina una a una después de que cada sesión haya terminado. De esta forma, se puede especificar en las reglas de Snort que sólo busque patrones en paquetes que pertenezcan a conexiones establecidas y, por tanto, falsos ataques como los realizados por aplicaciones similares a *stick* podrían ser evitados, ya que en estos casos los paquetes que se envían a la red no suelen pertenecer a ninguna sesión establecida. De esta forma no se sobrecargaría Snort y no se llegaría al punto de tener que omitir paquetes del análisis.

Otra ventaja de que Snort esté orientado a conexión es que pueden detectarse escaneos producidos fuera de secuencia, como por ejemplo el escaneo *stealth FIN*, un método de escaneo usado por Nmap. De acuerdo con el protocolo TCP, los paquetes FIN sólo aparecen durante la secuencia de fin de conexión, como se muestra en la figura 4.7. Por tanto, si un paquete FIN se envía hacia un puerto que no tiene abierta ninguna conexión, el servidor debería responder al cliente con un paquete RST (reset connection). En este caso, el preprocesador Stream5 enviará una alerta advirtiendo de un paquete FIN enviado hacia una sesión que nunca llegó a establecerse. Será tarea del administrador estudiar esta alerta y tomar las medidas oportunas contra el atacante.

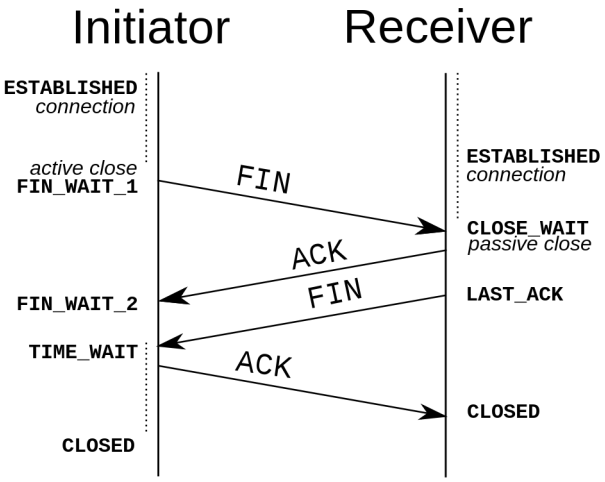

Figura 4.7: Fin de conexión en el protocolo TCP. Fuente: Wikipedia.org.

4.3.3. Motor de detección

Una vez que los paquetes han sido preprocesados llega el momento de comprobar si contienen posibles ataques. Para ello, el motor de detección toma cada paquete que recibe del preprocesador y compara el campo de datos de cada uno con el conjunto de reglas del IDS. Si se produce alguna coincidencia entonces se envía el paquete hacia al módulo de salida de Snort, que se encargará de emitir una alerta y de generar los logs correspondientes. En la figura 4.8 se muestra un ejemplo gráfico del funcionamiento de este componente.

Todos y cada uno de los componentes de Snort juegan un papel importante e imprescindible en el correcto y eficiente funcionamiento de la aplicación. No obstante, no hay que olvidar que el objetivo principal de Snort es detectar posibles ataques. Por tanto, el motor de detección es

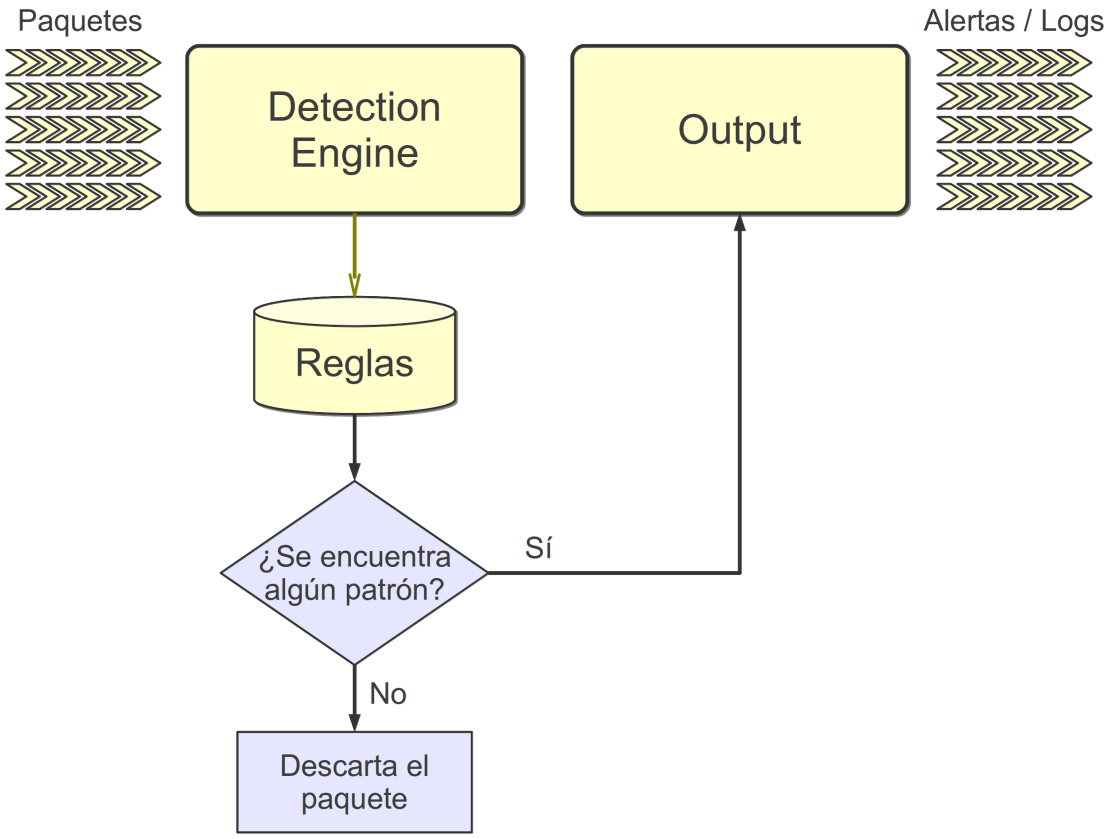

Figura 4.8: Funcionamiento del motor de detección.

considerado el componente central del IDS sobre el que girarán el resto de componentes. Es más, se podría decir que cada uno de los componentes anteriores realiza su labor con vistas a aportar más información al motor de detección. A su vez, éste será el componente más complejo y extenso de Snort. Su estudio y comprensión se encuentran a un nivel de dificultad por encima del resto.

Cualquier IDS que, como Snort, tome decisiones en función a la base de firmas cuenta con varios conjuntos de reglas, también conocidos como *rulesets*, que están agrupados por categoría y se actualizan con frecuencia. Esta agrupación se basa en el tipo de ataques, como pueden ser troyanos y buffer overflows, y en el tipo de aplicación atacada, como por ejemplo telnet, ftp o ssh. A continuación, se dedica un punto de este capítulo al estudio de las reglas.

4.4. Reglas

Las reglas son una parte fundamental de cualquier IDS y de ellas depende el nivel de protección de un sistema. No importa la calidad del IDS que se implante si éste no viene acompañado de un conjunto de reglas que proporcionen una buena seguridad en el sistema.

Snort cuenta con un conjunto de reglas oficiales que son actualizadas frecuentemente y que pone a disposición de sus usuarios de manera gratuita. Estas reglas son desarrolladas por el equipo de

Sourcefire, empresa fundada por Martin Roesch que continuó con el desarrollo de Snort desde el año 2003. A este conjunto de reglas se las denomina reglas VRT (Vulnerability Research Team™) y Sourcefire se encarga también de gestionarlas, añadiendo, modificando o eliminándolas en cada versión nueva.

También existen conjuntos de reglas desarrolladas por otros organismos, como Emerging Threats, que pone a disposición de los usuarios de Snort y Suricata conjuntos de reglas gracias al respaldo de una comunidad Open Source.

No obstante, ya sabemos que cada usuario, en función de sus necesidades, puede personalizar los conjuntos de reglas (*rulesets*), deshabilitando alguno de ellos, eliminando reglas concretas o incluso creando sus propias reglas.

Estos conjuntos de reglas que usará Snort se seleccionan antes de la ejecución de Snort para que cuando la aplicación arranque y comience a recibir paquetes de la red, estos puedan ser analizados en base a una estructura de reglas ya creada que contenga toda la información relativa al conjunto de firmas. Esta fase es muy importante para que el análisis se realice lo más rápido posible. Para entenderlo mejor:

- primero se describirán los campos y opciones principales que contiene una **regla** para, de esta forma, comprender cómo se crea una regla personalizada,
- en segundo lugar se explicará **cómo se forma la estructura de reglas** en base a los conjuntos de reglas leídos en el arranque de Snort y
- en tercer y último lugar se describirá el **camino recorrido por un paquete** al pasar por el motor de detección y cómo se decide si contiene o no un posible ataque definido en las reglas.

4.4.1. Lenguaje de las reglas

Snort usa un lenguaje de descripción de reglas simple, flexible y bastante potente. La mayoría de las reglas están escritas en una única línea, esto se debe a que, hasta la versión 1.8, Snort no soportaba el uso de la barra invertida (\) para indicar un cambio de línea, por tanto las reglas no podían escribirse usando más de una línea. Se informa al lector que en adelante podrán aparecer algunos ejemplos en los que se use la barra invertida para indicar un cambio de línea en la regla.

Cada regla está dividida en dos secciones bien diferenciadas, la cabecera de la regla y sus opciones. La cabecera contiene la acción que se debe tomar en el caso de que un paquete sea detectado por dicha regla, el protocolo al que va dirigida dicha regla, las direcciones IP de origen y destino y sus máscaras y la información de los puertos de origen y destino. La sección de las opciones de la regla contiene, entre otros, los mensajes de alerta y la información que indica qué partes del paquete deberían ser inspeccionadas para determinar si se ha de lanzar la acción estipulada por la cabecera de la regla.

Como se muestra en el ejemplo siguiente, el texto de una regla que va desde el inicio hasta el primer paréntesis forma la cabecera de la regla, mientras que las opciones de la regla están descritas entre los paréntesis. Las opciones deben finalizar con el carácter punto y coma (`;`), mientras que para

indicar los argumentos de una opción debe escribirse el carácter dos puntos (`:`) después de la opción y situar los argumentos a continuación, separando cada uno con una coma (`,`).

```
alert udp any any -> 192.168.1.1/24 23 \
            (msg:"telnet"; content:"connect"; isdataat:1,relative;)
```

El primer elemento que aparece en la **cabecera de una regla** es la acción regla (*rule action*), que indica qué se debe hacer con el paquete en el caso de que se detecte como amenaza. Existen cinco tipos de acciones que pueden producirse en Snort que se enumeran y describen a continuación:

- `alert`: genera una alerta y después registra el paquete.
- `log`: únicamente registra el paquete.
- `pass`: ignora el paquete
- `activate`: genera una alerta y activa otra regla (regla dinámica).
- `dynamic`: permanece inactiva hasta que una regla de tipo *activate* la activa. Una vez activada funciona como una regla de tipo *log*.
- `drop`: bloquea el paquete y lo registra como si fuera una regla *log*.
- `reject`: bloquea el paquete, lo registra y envía un paquete RST, si se trata de un protocolo TCP, o un mensaje de puerto inalcanzable ICMP, si se trata de un protocolo UDP.
- `sdrop`: bloquea el paquete pero no lo registra.

El siguiente elemento representa el protocolo. Actualmente, Snort analiza 4 tipos de protocolos: TCP, UDP, IP e ICMP.

Los cinco siguientes elementos representan el camino de la comunicación. El primero de ellos es la dirección IP y máscara de origen, mientras que el segundo representa el puerto de origen. El tercer elemento indica la dirección de la comunicación, siendo posibles dos operadores, el que representa un único sentido (`->`) y el que muestra sin embargo doble sentido (`<>`), también conocido como comunicación bidireccional. Los dos últimos elementos representan la dirección IP y el puerto del destino de la comunicación.

En los campos IP y puerto puede utilizarse el operador `any` para referirse a cualquier comunicación posible. Es decir, para esa regla se buscará en paquetes con cualquier dirección IP y puerto de origen y destino.

```
alert tcp any any -> any any
```

En los elementos relativos a las direcciones IP, tanto origen como destino, pueden definirse listas de direcciones IP mediante el uso de los caracteres corchetes (`IP1,IP2]`) y

```
alert tcp any any -> [192.168.1.0/24,10.1.1.0/24] any
```

evitar determinadas IPs del análisis usando el carácter de admiración (`!IP`).

```
alert tcp any any -> !192.168.1.0/24 any
```

Para los elementos que representen puertos, puede usarse el carácter de dos puntos (`:`) para especificar rangos de puertos y

```
alert udp any any -> any 1:1024
```

el carácter admiración (`!`) para evitar determinados puertos,

```
alert tcp any any -> any !23
```

como en el caso de las direcciones IPs.

En cuanto a las **opciones de una regla**, forman cuatro grandes categorías: general, payload, non-payload y post-detection. A continuación se explicarán estas cuatro categorías y se describirán las opciones más importantes de cada una:

- General: Formada por opciones que proporcionan información acerca de la regla pero que no tienen efecto durante la fase de detección.
  - `msg`: Especifica el mensaje que debe mostrarse en pantalla o ser registrado en el fichero de log cuando se haya detectado un paquete como posible amenaza.
  - `gid`: Identifica qué parte de Snort ha generado el evento. Si no se especifica manualmente Snort asignará por defecto un 1 a este valor.
  - `sid`: Identificador único de regla. Identifica una regla inequívocamente del resto.
  - `rev`: Indica el número de versión de la regla.
- Payload: Las opciones de esta categoría sí se tendrán en cuenta en la fase de detección. En concreto, harán referencia al payload del paquete.
  - `content`: Este campo es uno de los más importantes de Snort. Permite al usuario crear reglas que busquen un patrón específico dentro del payload del paquete. Cuando el motor de detección se disponga a comprobar la existencia de este patrón en el payload de un paquete, invocará al algoritmo de búsqueda de patrones (*fast pattern matcher*) que se haya indicado en el archivo de configuración de Snort. Previamente, se habrá creado su estructura de detección en el proceso de arranque de Snort. Si se encuentra el patrón en cualquier parte del payload entonces se ejecutarán el resto de opciones. Existen gran cantidad de modificadores para este campo. A continuación se mostrarán los más útiles e importantes que Snort pone a disposición:
    - `nocase`: Este modificador indica a Snort que debe realizar la búsqueda del patrón sin hacer distinción entre mayúsculas y minúsculas.
    - `depth`: Seguido de un argumento entero, especifica hasta cuantos bytes del payload deben ser leídos en el proceso de búsqueda de patrones. Sólo se permiten un valor mayor o igual a la longitud del patrón, siendo 1 el valor mínimo y 65535 el máximo.
    - `offset`: Permite desplazar el punto de inicio del payload en el que comenzará la búsqueda de patrones. El número de bytes del desplazamiento vendrá indicado por el argumento entero del modificador. Usado en combinación con el modificador

anterior se pueden realizar búsquedas sobre fragmentos concretos del payload.

- `distance`: El valor entero que tome este modificador corresponderá con el número de bytes que deberán ser ignorados desde que se encuentre un patrón, hasta que se deba iniciar la búsqueda del siguiente.
- `within`: En un contexto de dos patrones a buscar en el payload, como en el caso anterior, el valor numérico que tome este modificador corresponderá al número máximo de bytes que deben pasar desde que se encuentra el primer patrón hasta que se pueda encontrar el siguiente.
- `http_client_body`: Restringe la búsqueda del patrón al cuerpo de una petición de cliente de tipo HTTP (HTTP client request).
- `http_uri`: Restringe la búsqueda del patrón al campo URI ya normalizado por el preprocesador HTTP Inspect.
- `http_raw_uri`: Al igual que el anterior, realiza la búsqueda del patrón en el campo URI, pero con la diferencia de que se realiza sobre la URI no normalizada.
- `fast_pattern`: Cuando aparece más de un campo `content` en la regla, únicamente se pasa el patrón de mayor longitud por el *fast pattern matcher*. Sin embargo si se prefiere que sea otro patrón el que pase por el *fast pattern matcher*, deberá añadirse este modificador detrás de su campo `content`.

◦ `urilen`: En este campo especifica la longitud exacta, mínima, máxima o el rango de longitud del URI en el que se quiere realizar la búsqueda.

◦ `isdataat`: Comprueba que el payload contiene un dato en la posición especificada por el argumento.

◦ `pcre`: Permite escribir reglas usando expresiones regulares compatibles con perl.

- Non-payload: Al igual que en el grupo anterior, también se tendrán en cuenta estas opciones en la fase de detección, pero esta vez no harán referencia al payload del paquete.

◦ `ttl`: Esta opción se usa para comprobar el valor del tiempo de vida (time-to-live) de un paquete. Pudiéndose especificar valores concretos o rango de valores.

◦ `tos`: Usada para comprobar el tipo de servicio (type-of-service) del paquete en cuestión. Usa argumentos de tipo entero y permite la negación de los mismos usando el carácter de admiración (`!`).

◦ `id`: Comprueba que el campo ID del paquete contenga un determinado valor. Algunas herramientas como escáneres y aplicaciones maliciosas usan ciertos identificadores por defecto, como el 31337.

◦ `ipopts`: Sirve para comprobar si están presentes ciertas opciones de la cabecera IP, como `sec` (IP Security), `nop` (No Op) o `ts` (Time Stamp).

◦ `dsize`: Se usa para comparar el tamaño del payload del paquete. Esta opción puede utilizarse para buscar tamaños de paquetes sospechosos, que es útil en la detección de ataques de tipo buffer overflows.

◦ `flags`: Comprueba la presencia de banderas específicas del protocolo TCP, como `SYN`,

`ACK`, `RST` o `FIN`.

- `flow`: Esta opción está estrechamente ligada al uso del preprocesador Stream5, el cual tenía la misión de hacer de Snort una aplicación orientada a conexión. En concreto, el uso de esta opción permite que la regla se aplique únicamente a ciertas direcciones del flujo de datos de una conexión establecida. Por ejemplo, si únicamente interesa aplicar la regla al flujo de datos que sale del cliente, se usaría como argumento la opción `from_client`. Otras opciones que podrían usarse como argumentos serían: `to_cliente`, `to_server`, `from_server`, `stateless`, `not_established`, `established`, etc.
- `seq`: Esta opción permite comprobar si el paquete tiene el número de secuencia que se indica como argumento.

- Post-detection: Estas opciones sólo se tienen en cuenta una vez detectada una amenaza, por lo que no influye en la fase de detección.
  - `logto`: Indica a Snort que las alertas producidas se registren en un fichero de log especial. El nombre del fichero se indica como parámetro. Esta opción es bastante útil si quieren aislarse diferentes tipos de ataques para estudiarlos por separado.
  - `activates`: Permite añadir una regla para que se active cuando se produzca un evento provocado por la regla actual.

4.4.2. Estructura de reglas

Una vez que se decide qué reglas van a ser usadas por el IDS, llega el momento de ejecutar la aplicación. Ésta leerá todas y cada una de las reglas, verificará que tienen una sintaxis correcta y descartará las que no sean válidas o no estén bien formadas. Las reglas que obtengan el visto bueno serán las que se utilizarán finalmente durante la ejecución.

Para conseguir que la aplicación sea lo más eficiente posible, todas y cada una de las reglas se almacenarán en una lista enlazada múltiple, denominada por algunos usuarios y desarrolladores de la comunidad como lista enlazada en 3D. En la figura 4.9 se representa el concepto de lista enlazada en tres dimensiones. Su construcción se ha basado en las reglas descritas en el texto 1. Este conjunto de reglas que se expone como ejemplo han sido extraídas del conjunto de reglas VRT que Snort pone a disposición de sus usuarios registrados.

La organización de las reglas que se lleva a cabo en la lista enlazada múltiple se clasifica en varios niveles que se explican a continuación:

1. El primer nivel de la lista pertenecerá al primer campo de la cabecera de la regla, es decir, al que indica la acción a seguir en el caso de que la regla produzca un evento. Se recuerda que las opciones posibles eran alert, log, pass, drop, activate, etc.
2. El segundo nivel corresponde al tipo de protocolo que es analizado, es decir TCP, UDP, ICMP o IP. Tanto el primer nivel como el segundo formarían la primera dimensión de la lista enlazada.

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any \
       (msg:"SHELLCODE JavaScript var shellcode"; flow:to_client,established; \
       content:" shellcode"; nocase;)
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any \
       (msg:"SHELLCODE JavaScript var heapspray"; flow:to_client,established; \
       content:" heapspray"; nocase;)
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any \
       (msg:"SHELLCODE Possible heap spray attempt"; flow:to_client,established; \
       content:"this.mem = new Array|28 29|";)

alert tcp $EXTERNAL_NET any -> $HOME_NET 143 \
       (msg:"IMAP Qualcomm WorldMail IMAP Literal Token Parsing Buffer Overflow"; \
       flow:to_server,established; dsize:>668;)
alert tcp $EXTERNAL_NET any -> $HOME_NET 143 \
       (msg:"IMAP MailEnable IMAP Service Invalid Command Buffer Overlow LOGIN"; \
       flow:to_server,established; content:"login|20 7B|"; depth:7; offset:3;)

alert tcp $HOME_NET any -> $EXTERNAL_NET 25 \
       (msg:"SPYWARE-PUT Keylogger gurl watcher runtime detection"; \
       flow:to_server,established; content:"X-Mailer|3A| GURL Watcher"; \
       fast_pattern:only;)
alert tcp $HOME_NET any -> $EXTERNAL_NET 25
       (msg:"SPYWARE-PUT Snoopware pc acme pro runtime detection"; \
       flow:to_server,established; content:"Attached file is PC Acme report"; \
       fast_pattern:only;)


alert udp $EXTERNAL_NET any -> $HOME_NET 5632 \
       (msg:"POLICY PCAnywhere server response"; content:"ST"; depth:2;)


alert ip $EXTERNAL_NET any -> $HOME_NET any \
       (msg:"SHELLCODE x86 inc ebx NOOP"; content:"CCCCCCCCCCCCCCCCCCCCCCCC";)
alert ip $EXTERNAL_NET any -> $HOME_NET any \
       (msg:"SHELLCODE x86 inc ecx NOOP"; content:"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";)

alert ip any any -> any any \
       (msg:"ATTACK-RESPONSES id check returned root"; content:"uid=0|28|root|29|";)


activate icmp $HOME_NET any -> $EXTERNAL_NET any \
       (msg:"ICMP Address Mask Reply"; icode:0; itype:18; activates:1;)

activate icmp $EXTERNAL_NET any -> $HOME_NET any \
       (msg:"ICMP PING"; icode:0; itype:8; activates:2;)
activate icmp $EXTERNAL_NET any -> $HOME_NET any \
       (msg:"ICMP Address Mask Reply undefined code"; icode:>0; itype:18; activates:1;)
activate icmp $EXTERNAL_NET any -> $HOME_NET any \
       (msg:"ICMP Address Mask Request"; icode:0; itype:17; activates:3;)
```

Texto 1: Ejemplo real de firmas extraídas de la base de firmas VRT de Snort.

3. Del nivel anterior nace otra lista enlazada conocida como la segunda dimensión, que contiene las estructuras RTN (Rule Tree Node). Esta estructura guarda la información de las direcciones IP y puertos de origen y destino.

4. El último nivel y tercera dimensión corresponde a las estructuras OTN (Option Tree Node), que parten a raíz de cada RTN. En esta estructura se almacenan las opciones de las reglas.

A modo de resumen, para formar la lista enlazada 3D desde el inicio hasta la lista de RTNs se usarán únicamente las cabeceras de las reglas. Después, de cada RTN colgará una lista de OTNs que representarán las opciones de cada regla. Por tanto, en una lista de OTNs todas y cada una de las reglas compartirán la misma cabecera.

En el ejemplo, las cabeceras usadas para formar la estructura hasta la lista de RTNs son las

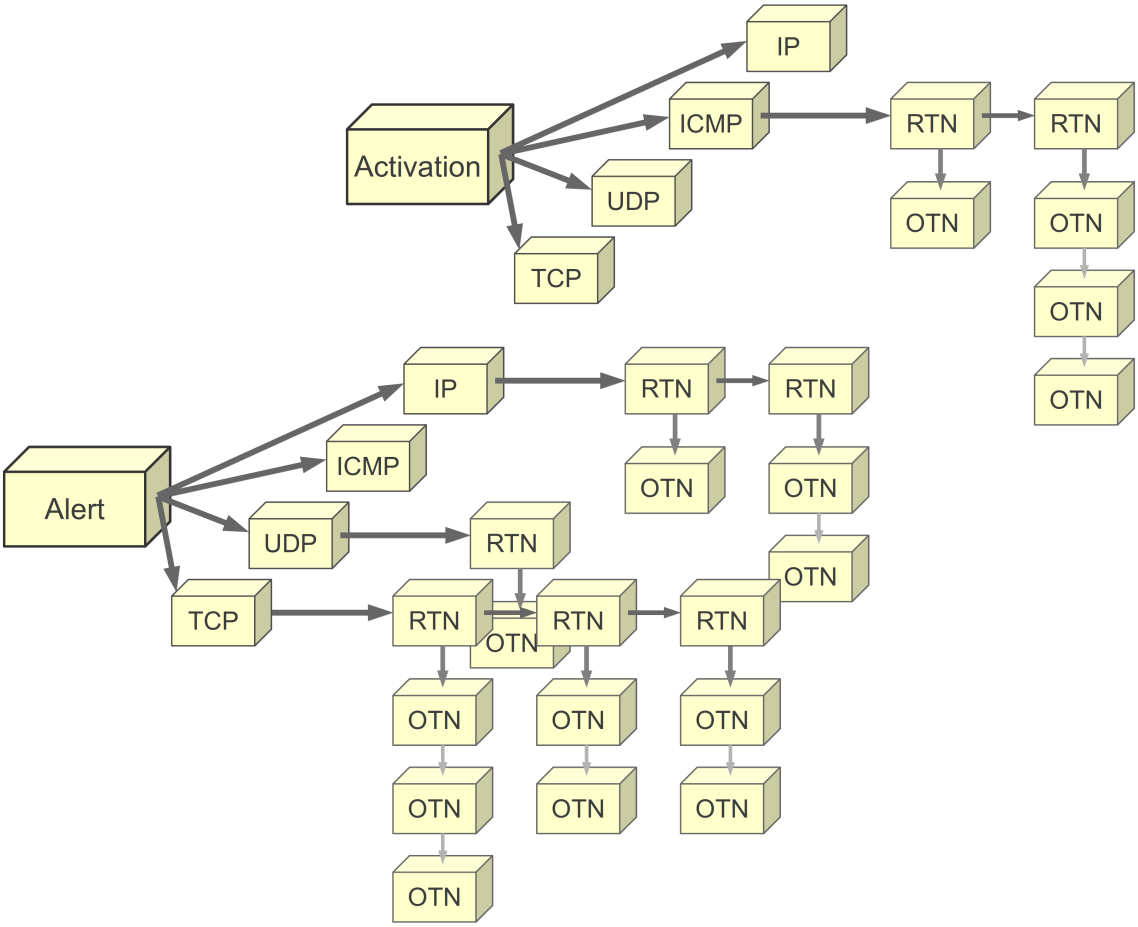

Figura 4.9: Esquema de lista enlazada en tres dimensiones.

siguientes:

- `alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any`
- `alert tcp $EXTERNAL_NET any -> $HOME_NET 143`
- `alert tcp $HOME_NET any -> $EXTERNAL_NET 25`
- `alert udp $EXTERNAL_NET any -> $HOME_NET 5632`
- `alert ip $EXTERNAL_NET any -> $HOME_NET any`
- `alert ip any any -> any any`
- `activate icmp $HOME_NET any -> $EXTERNAL_NET any`
- `activate icmp $EXTERNAL_NET any -> $HOME_NET any`

En la figura 4.10 se muestra un ejemplo de lista enlazada construida tomando las reglas de tipo `alert tcp` del texto 1.

4.4.3. Camino recorrido

En el momento que finaliza la construcción de la estructura de reglas se puede iniciar la recepción de paquetes y su posterior análisis. Dada la estructura creada, el camino que tomará un paquete que

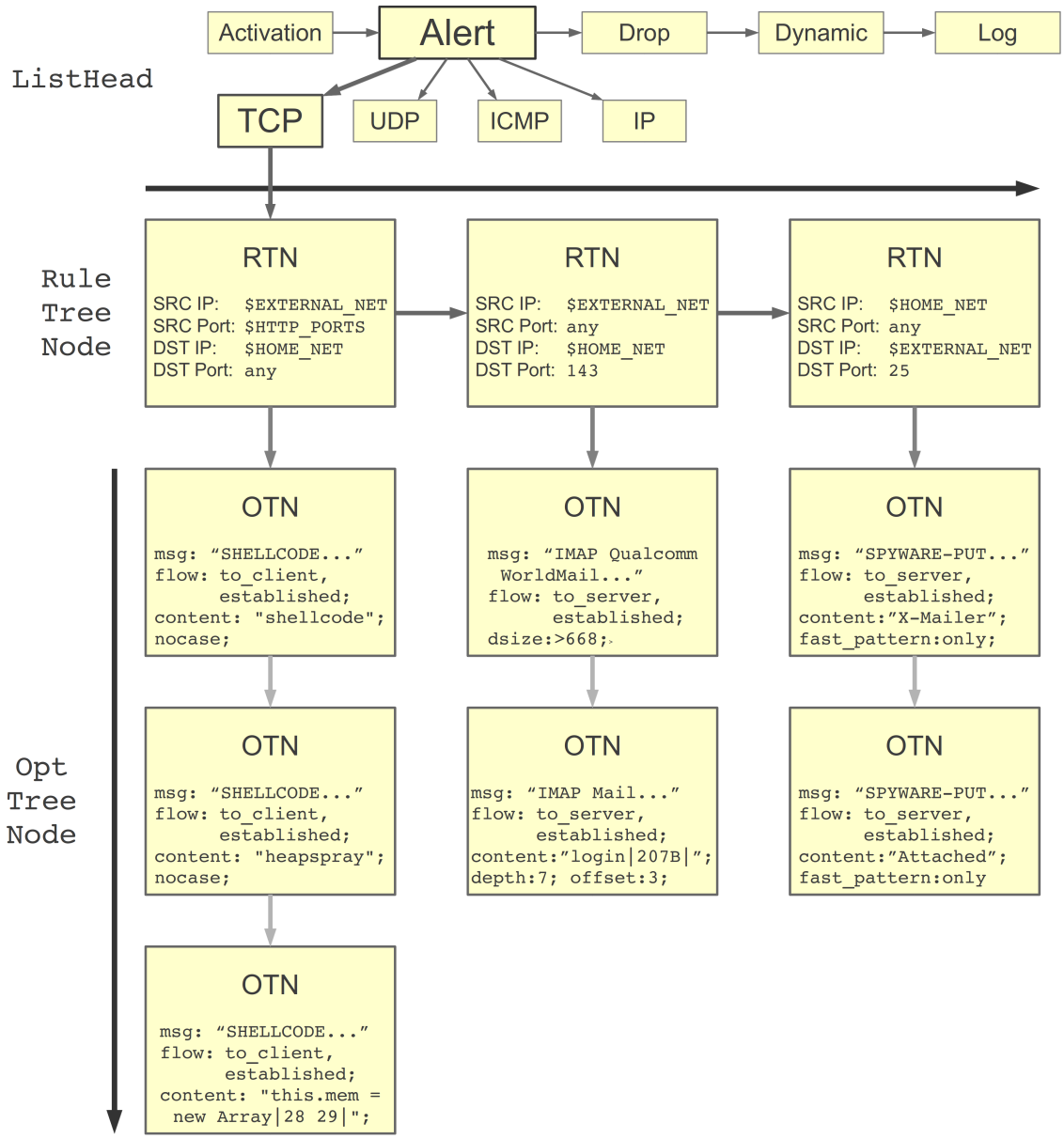

Figura 4.10: Esquema de lista enlazada de reglas.

deba recorrerla es el siguiente:

1. Al llegar el paquete, éste deberá recorrer la `ListHead`, que almacena las reglas en base a la acción a tomar por cada una. En dicha lista habrá tantos elementos como tipos de acciones hayan en el conjunto de reglas. Por tanto, si únicamente hubiera un tipo de acción en todo el *ruleset* habría un único elemento en la `ListHead`, si existiera más de uno el paquete tendría que recorrer la lista elemento a elemento.
2. Una vez que el paquete se sitúa en un elemento, éste seguirá una bifurcación, de las cuatro posibles, en función del protocolo que se trate. Como se muestra en el ejemplo de la figura 4.10, si el protocolo es TCP el paquete seguirá el camino marcado en la imagen hasta llegar a la lista de RTNs.

3. Esta lista se recorrerá completamente deteniéndose en los `RTN` que coincidan con los campos correspondientes a las direcciones IP y puertos de origen y destino. Por ejemplo, si el paquete procede de una dirección IP externa (`$EXTERNAL_NET`), viaja a través del puerto 1234 y tiene como destino una dirección IP que pertenece a `$HOME_NET` con puerto 143, entonces se detendrá en el segundo `RTN` del ejemplo. No queda ahí la cosa, puesto que si además el puerto 1234 pertenece a `$HTTP_PORTS`, que es definido por el usuario, entonces no sólo se detendrá en el segundo `RTN`, sino que lo hará también en el primero.
4. Por último, una vez detenido en un `RTN`, este deberá pasar por cada una de las estructuras `OTN` de la lista para verificar si el paquete cumple con las opciones de alguna regla.

Aunque se verá en el siguiente capítulo, es importante indicar en este punto una de las peculiaridades de la búsqueda de patrones en Snort.

Hasta ahora, ha sido lógico suponer que se crearía un único DFA para la búsqueda de patrones en base a todo el conjunto de reglas de Snort, pero no es así. Después de comprender cómo funciona la estructura organizativa de las reglas dentro de Snort, se puede entender mejor cómo se realiza verdaderamente la búsqueda de patrones.

Continuando la explicación anterior del camino seguido por un paquete, la que se divide en cuatro pasos, los próximos párrafos podrían ser considerados como una extensión del cuarto paso que sigue dicho paquete en su camino.

Una vez que se llega a una estructura RTN, de ésta cuelgan un cierto número de OTNs, cada una correspondiente a una regla. Pueden haber OTNs que incluyan campos de tipo `content`, o no. Si al menos existe una que lo incluya entonces en este caso Snort habrá creado por cada RTN, antes de iniciar la recepción de paquetes, un DFA formado por los patrones de los campos `content` de cada regla que pertenezca a un mismo RTN. Por tanto, se podrían llegar a construir tantos DFAs como RTNs hubiera dentro de la lista enlazada.

Rescatando las reglas de ejemplo del texto 1, se ha visto que se crearon ocho RTNs, pero no se ha mencionado nada respecto a cuáles de ellos contienen un DFA. Si nos fijamos en las reglas, sólo hay seis RTNs con al menos una regla que incluya el campo `content`, esto significa que únicamente esas seis estructuras RTNs contendrán un DFA para la búsqueda multi-patrón.

Por tanto, esta forma de estructurar las reglas también mejora el rendimiento de la búsqueda de patrones que se produce en el motor de detección de Snort puesto que se sabe, según los anteriores capítulos, que el rendimiento de un algoritmo de búsqueda de patrones comienza a verse afectado a medida que se incrementa el número de firmas. Y precisamente, de este modo se reduce el número de patrones que formarían el DFA.

Para entenderlo mejor se plantea un símil que sirve de ejemplo. Suponga el lector que se tiene un sistema de búsqueda de patrones cuyo diccionario lo forman palabras normales y los elementos en los que se debe realizar la búsqueda lo forman libros con portadas de distintos colores, habiendo únicamente 4 colores posibles.

Para este sistema, la regla está formada por el color del libro en el que se desea buscar y la palabra que quiere localizarse. También, suponemos un conjunto de 100 reglas y suponemos también que

habiendo dividido las reglas en grupos por colores, se obtengan 4 grupos de 25 reglas cada una. Llegado un momento, comienzan a llegar libros al sistema y se debe realizar la búsqueda de patrones. Ante esta situación se podría actuar de dos formas diversas:

- La primera sería crear una único DFA formado por los 100 patrones y pasar cada libro por dicho DFA. Cuando se produjera una coincidencia se comprobaría el color del libro y se decidiría si se ha cumplido la regla o no.
- La segunda opción sería crear cuatro estructuras DFA, cada una de ellas formada por los 25 patrones extraídos de las 25 reglas de cada color. Por tanto, a la llegada de un libro, se comprobaría el color y se redirigiría al DFA que le correspondiera. Al encontrarse una palabra se sabría con seguridad que se ha cumplido con la regla.

En la primera situación se debería realizar la comprobación del color tantas veces como palabras aparecieran en el libro, sin embargo en la segunda únicamente se realizaría esta comprobación una vez, al inicio.

En lo que se refiere al rendimiento de un DFA de 100 o de 25 patrones, la teoría dice que no afecta el número de patrones en el rendimiento y que, por tanto, ambos contarán con el mismo rendimiento. No obstante, ya se ha visto que la cantidad de memoria usada por un DFA crece exponencialmente a medida que aumenta el número de patrones usados. Este consumo de memoria, en un sistema ideal no debería afectar al rendimiento, pero sabemos que en un sistema real sí que lo hace. Por tanto, normalmente el rendimiento del DFA en la segunda situación será mayor.

Puesto que la segunda situación será más favorable en cuanto a rendimiento, se puede llegar a la conclusión de que es conveniente realizar una división del DFA siempre que sea posible, y esto es precisamente lo que hace Snort.

Esta forma que tiene Snort, tanto de organizar las reglas, como de "catalizar" los DFAs, cambian de manera importante las expectativas que se tenían respecto a la implementación del filtro sigMatch, puesto que en los resultados que obtuvo el equipo de Jignesh M. Patel se constató un descenso de la mejora que proporcionaba el filtrado a medida que disminuía el número de firmas del sistema de búsqueda de patrones en el que se quería implantar. Por tanto, si Snort cuenta con un grupo de unas 5.500 firmas, realmente se estaría contando con multitud de grupos de un número de firmas bastante menor.

4.5. Fichero de configuración de Snort: snort.conf

Durante todo el documento se ha hablado mucho de este archivo, en él residen todas las opciones configurables de Snort, como son la declaración de variables de red, las opciones de configuración del descodificador, del motor de detección, del preprocesador y de los módulos de salida, la configuración personalizada del conjunto de reglas, etc.

Cualquier administrador de sistemas que trabaje con Snort debería estar familiarizado con las principales opciones de este fichero o, al menos, conocer cómo se organiza por dentro.

En este punto se describirán brevemente las partes que forman dicho fichero y se explicarán con más detalle los apartados que resulten de mayor interés y estén relacionados con el desarrollo del proyecto. Se exponen a continuación:

1. Configuración de las variables de red: En este paso se establecen las variables que después se utilizarán en los campos IP y puerto de cada regla, como por ejemplo `HOME_NET` o `SSH_PORTS`. Se definirían del siguiente modo:

```
# Setup the network addresses you are protecting
ipvar HOME_NET 192.168.1.0

# List of ports you want to look for SSH connections on:
portvar SSH_PORTS 22
```

2. Configuración del descodificador.
3. Configuración del motor de detección: Es en este punto donde se elige el método de búsqueda de patrones que se usará al ejecutar la aplicación. También se definen más opciones relacionadas, como pueden ser la longitud máxima de patrón o la inclusión de reglas any-any en un grupo aparte. Un ejemplo sería el siguiente:

```
# Configure the detection engine
config detection: search-method ac split-any-any max-pattern-len 20
```

4. Configuración de las librerías dinámicas.
5. Configuración del preprocesador: Entre otras opciones, en este apartado se pueden activar o desactivar módulos de preprocesador, así como establecer las opciones de cada uno por separado. En el ejemplo siguiente se activa el módulo de preprocesador Frag3 mientras que el módulo HTTP_Inspect permanece deshabilitado:

```
# Target-based IP defragmentation.
# For more information, see README.frag3
preprocessor frag3_global: max_frags 65536
preprocessor frag3_engine: policy windows detect_anomalies \
      overlap_limit 10 min_fragment_length 100 timeout 180

# HTTP normalization and anomaly detection.
# For more information, see README.http_inspect
# preprocessor http_inspect
# preprocessor http_inspect_server
```

6. Configuración de los módulos de salida.
7. Creación personalizada del conjunto de reglas: En esta sección se incluyen ficheros de reglas al conjunto de reglas final. Estos ficheros están organizados por tipos de ataques. En el ejemplo se muestra la inclusión del fichero de reglas de tipo Denial-of-service (DoS) al conjunto final de reglas, mientras que no se incluyen las de tipo DNS:

```
include $RULE_PATH/dos.rules
# include $RULE_PATH/chat.rules
```

8. Creación personalizada del conjunto de reglas del preprocesador y del descodificador.

9. Creación personalizada de las reglas de tipo Share Object (SO).

Esta explicación será suficiente para entender mejor tanto lo que queda de capítulo, como ciertos apartados de los siguientes capítulos, como por ejemplo el proceso de integración del filtrado en Snort, el cual deberá poder ser habilitado y deshabilitado desde el fichero de configuración, para lo que habrá que modificar algunos archivos del código fuente relacionados con la lectura del fichero `snort.conf`.

4.6. Dentro de Snort

Hasta ahora se ha dedicado el capítulo a explicar el funcionamiento de Snort, sin profundizar demasiado en su código fuente. En este punto se hará un análisis más a fondo de la aplicación, indicando dónde se localizan los componentes más importantes y cómo se realizan ciertos procesos importantes.

El recorrido por el código se dividirá en dos partes bien diferenciadas. La primera de ellas será la fase de inicio de Snort, en la que se crean las variables y estructuras principales, se lee del fichero de configuración de Snort, se analiza el conjunto de reglas, se configuran los componentes y se crea la estructura de detección. En la segunda parte se reciben los paquetes de la red a través del módulo DAQ, se descodifican y se envían al preprocesador, para que posteriormente se redirijan al motor de detección de Snort, que será el encargado de realizar el análisis definitivo.

En la figura 4.11 se muestra un diagrama de secuencia de la ejecución de Snort diferenciada en dos partes; el diagrama situado a la izquierda muestra la primera fase que se ha explicado en el párrafo anterior, mientras que la segunda fase se corresponde con el diagrama situado a la derecha.

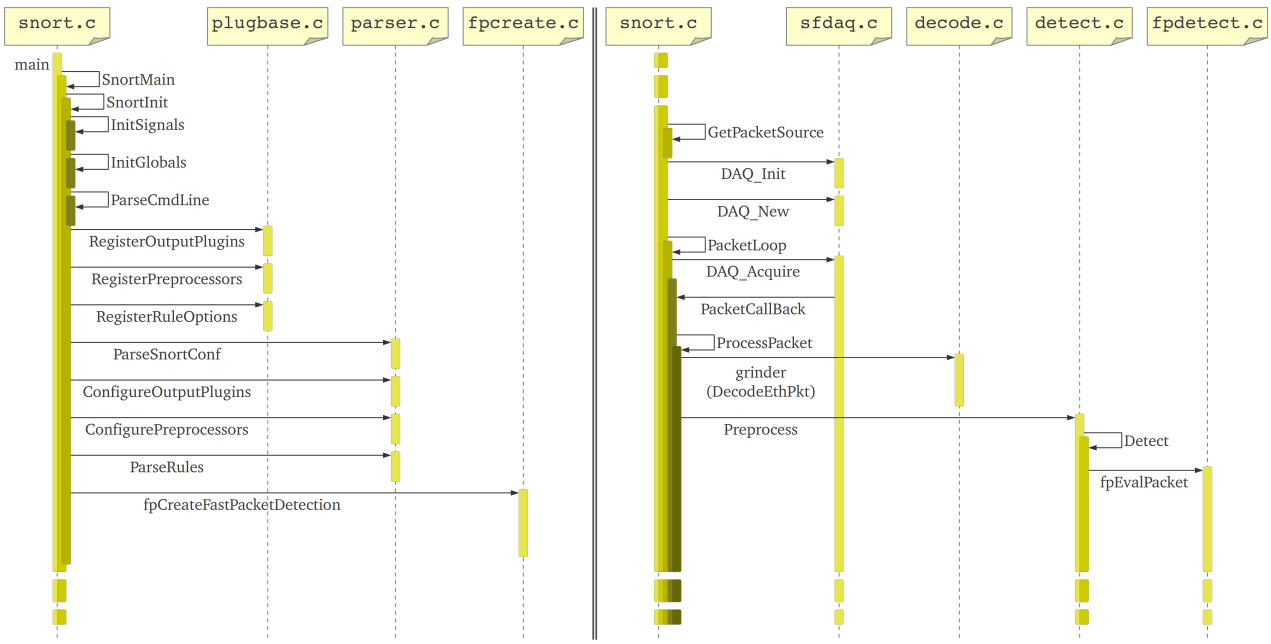

Figura 4.11: Diagramas de secuencia de las fases offline (izquierda) y online (derecha) llevadas a cabo en Snort.

Ya en el segundo capítulo se introdujeron los conceptos de fase *offline* y *online*. Si extendemos esta denominación al capítulo presente, podríamos denominar "fase *offline*" a la primera fase de ejecución de Snort y "fase *online*" a la segunda. No es muy aventurado elegir esta denominación puesto que, de acuerdo a lo que se definió en el segundo capítulo, la primera fase de Snort se encarga, entre otras cosas, de crear la estructura de detección en base al conjunto de reglas, mientras que una de las tareas de la segunda fase consiste en llevar a cabo la lectura los paquetes entrantes (texto) y su posterior análisis con ayuda de la estructura de detección ya creada.

Una vez aclarada esta última cuestión, se dará a una explicación más detallada de la fases *offline* y *online*. comenzando por la primera de ellas.

4.6.1. Fase offline

Esta fase comprende desde la ejecución de Snort hasta que se termina de crear la estructura de detección. En la figura 4.12 se muestra su diagrama de secuencia de ejecución, en el que aparece el hilo de funciones que se ejecutan y una breve descripción del objetivo de cada una.

A continuación, se detallará cada una de las funciones mostradas en la figura 4.12 en ese mismo orden[1]:

1. Al ejecutar Snort se invoca la función `main`, que a su vez ejecutará la función `SnortMain`, pasándole la información de los argumentos recogidos en la línea de comandos a través de las variables `argc` y `argv`.

2. Es la función `SnortMain` la que verdaderamente gestiona la ejecución de la aplicación. En la fase *offline* se envía el control directamente a la función `SnortInit`, pasando de nuevo los argumentos `argc` y `argv`.

3. La función `SnortInit` lleva a cabo prácticamente toda la gestión de la fase *offline*. En primer lugar inicializa las señales usadas por Snort, mediante la invocación a la función `InitSignals`, y algunas de las variables y estructuras globales necesarias para la aplicación, al ejecutar la función `InitGlobals`.

4. Una vez hecho esto, llama a la función `ParseCmdLine` para analizar sintácticamente los argumentos de la línea de comandos, para ello se les pasan los parámetros `argc` y `argv` que se recogieron en la función `main`.

5. Después, comienza el registro de los plug-ins de salida, los preprocesadores y las opciones de las reglas. Todo esto se realiza al invocar, respectivamente, a las funciones `RegisterOutputPlugins`, `RegisterPreprocessors` y `RegisterRuleOptions`.

6. Acto seguido, a través de la función `ParseSnortConf` se lleva a cabo el análisis del archivo de configuración de Snort (`snort.conf`), en el cual están descritas las opciones gestionables de Snort. En este archivo se declaran las variables de red, se configuran el descodificador, los preprocesadores, el motor de detección, los plug-ins de salida y las

1 *En el anexo I se muestran fragmentos del código fuente de las funciones más importantes involucradas en los procesos offline y online de Snort que se describen en este capítulo.*

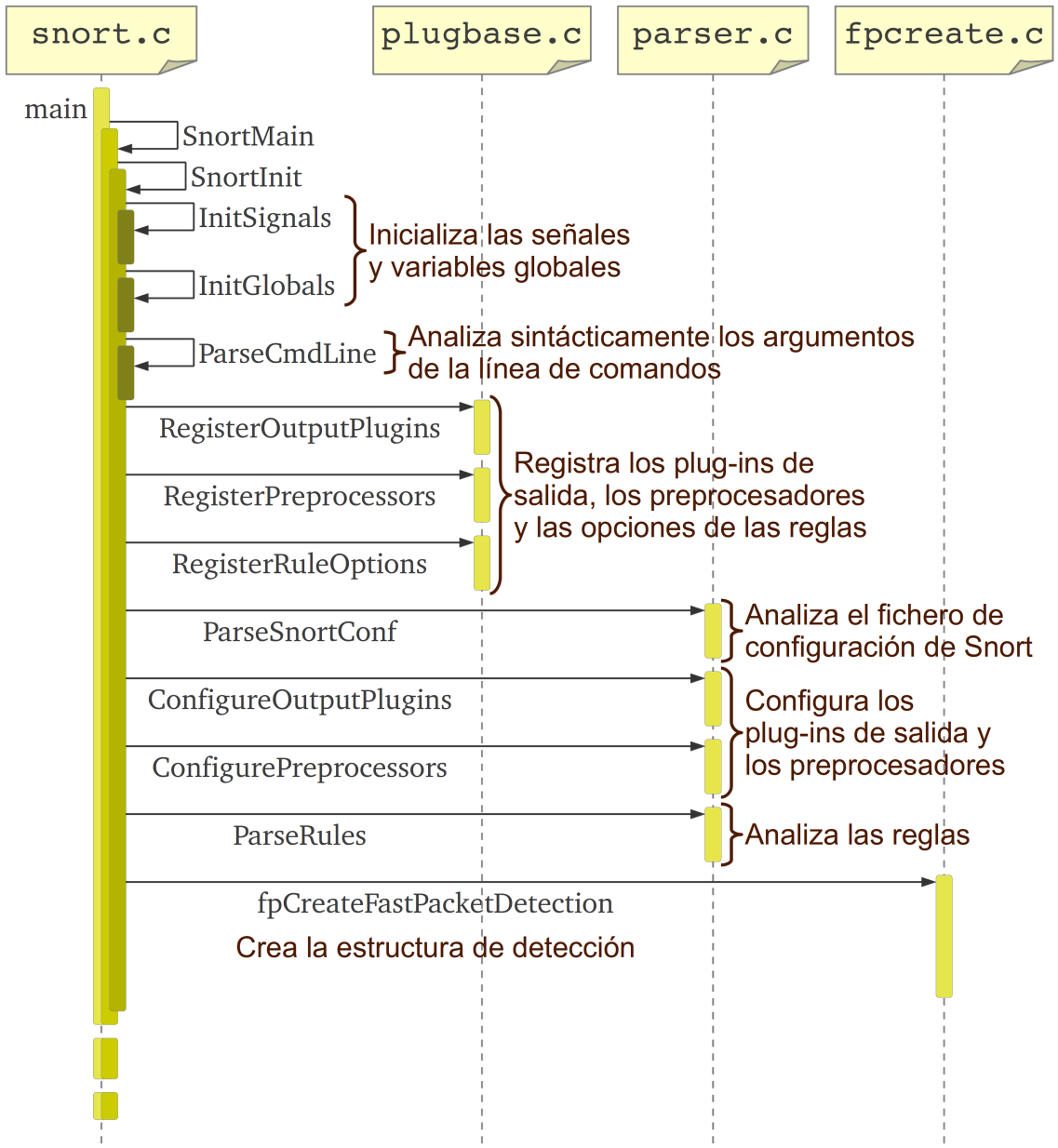

Figura 4.12: Diagrama de secuencia de la fase offline de Snort.

librerías dinámicas y se localizan los archivos que contienen el conjunto total de reglas.

7. Cuando ya se han almacenado todas las opciones del fichero de configuración en la estructura `SnortConfig`, situada en el archivo snort.h, se procede a la configuración de los plug-ins de salida y preprocesadores mediante las funciones `ConfigureOutputPlugins` y `ConfigurePreprocessors`, respectivamente.

8. Posteriormente se inicializan los plugins dinámicos y se configuran los preprocesadores dinámicos, esto último también a través de la función `ConfigurePreprocessors`. A continuación, mediante la función `ParseRules`, se analizan las reglas y se crea la lista enlazada compuesta de la información de todas y cada una de las reglas.

9. El último paso de la fase *offline* consiste en crear la estructura de detección mediante la ejecución de la función `fpCreateFastPacketDetection`, situada en el fichero `fpcreate.c`.

Esta última etapa merece especial atención puesto que es el punto de la fase *offline* que resulta más interesante de cara al desarrollo del proyecto de filtrado previo, por lo que se estudiará con más detalle. En la figura 4.13, se muestra un esquema que representa el recorrido que sigue la ejecución desde que la función es invocada hasta que se alcanza el motor de búsqueda multi-patrón (MPSE), el cual creará la estructura de detección.

El objetivo de la función `fpCreateFastPacketDetection` no es otro que el de crear grupos de patrones para la estructura de búsqueda multi-patrón. Paro ello se deben añadir estructuras de datos y ordenar la información de las reglas que ha obtenido previamente. Este proceso se lleva a cabo invocando al resto de funciones tal y como se muestra en la figura 4.13.

En concreto, se crearán estructuras `PORT_GROUP` a partir de estructuras `PortObject`, que están agrupadas por la estructura `rule_port_tables_t`. A continuación, se explicará, por orden de

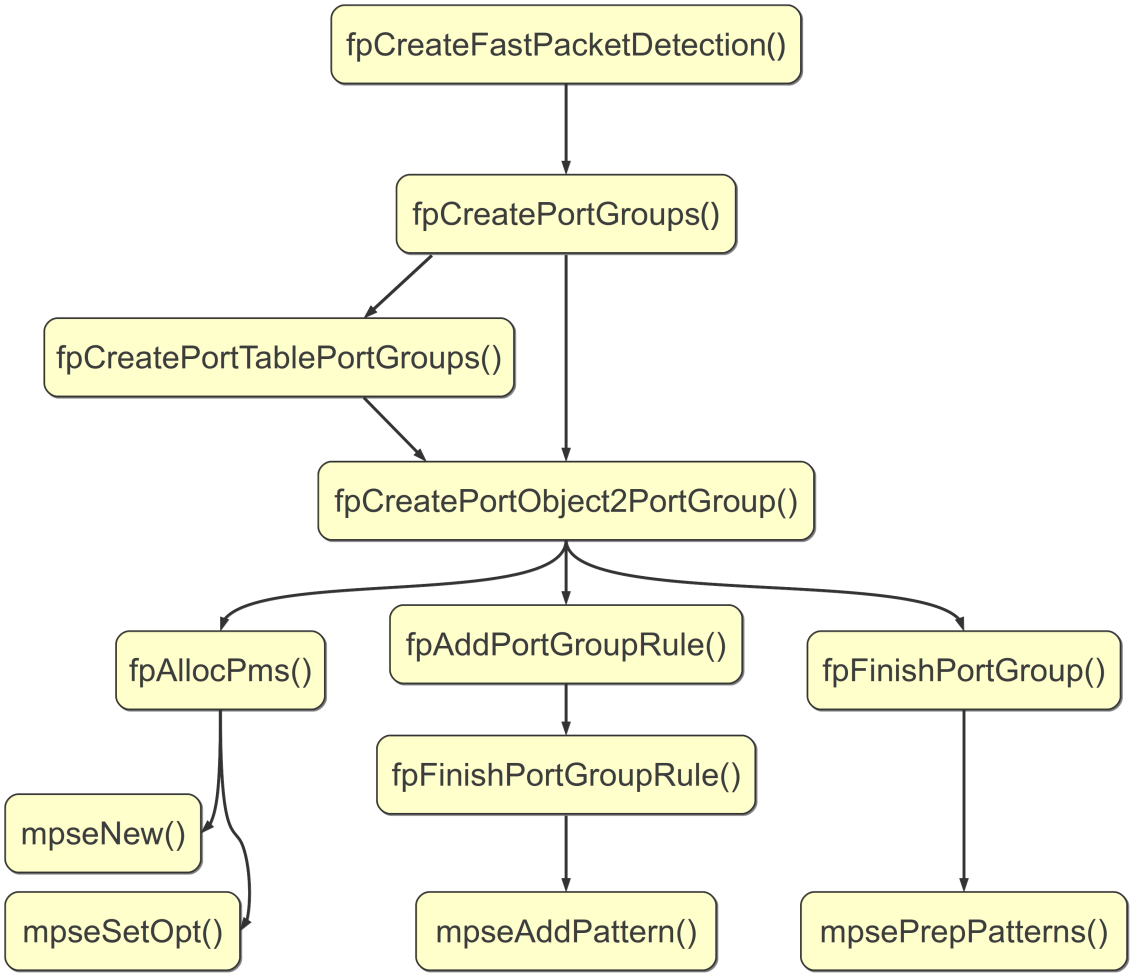

Figura 4.13: Esquema representativo del camino seguido desde la función fpCreateFastPacketDetection hasta el motor de búsqueda multi-patrón.

llamada cada una de las funciones que intervienen en este proceso:

1. La primera función de la secuencia a seguir es `fpCreatePortGroups`, a la que se le pasan por parámetro las estructuras `rule_port_tables_t` y `SnortConfig`. La creación de las estructuras `PORT_GROUP` continúa en esta función. Por cada tipo de protocolo se invoca dos veces a la función `fpCreatePortTablePortGroups`, una por puerto origen y otra por puerto destino, y una vez a la función `fpCreatePortObject2PortGroup`, para los casos en los que la pareja puerto origen y destino sea `any any`.

2. Cuando se llama a la función `fpCreatePortTablePortGroups`, esta termina invocando a `fpCreatePortObject2PortGroup`, que realiza la llamada al motor de búsqueda de patrones a través de `fpAllocPms`, `fpAddPortGroupRule` y `fpFinishPortGroup`, por este orden.

3. `FpAllocPms` ejecuta la función `mpseNew`, que crea la base de la estructura de detección en función de las opciones que se establecieron en el archivo de configuración de Snort, y la función `mpseSetOpt`, que activa la compresión de estados del DFA.

4. `FpAddPortGroupRule` invoca a `fpFinishPortGroupRule`, que a su vez ejecuta la función `mpseAddPattern`, encargada de añadir los patrones de las reglas que incluyan en las opciones el campo `content` a la estructura de detección del algoritmo que se eligió en el fichero de configuración.

5. `fpFinishPortGroup` ejecuta la función `mpsePrepPatterns`, la cual compila la estructura de detección en base al algoritmo seleccionado.

4.6.2. Fase online

Una vez construida la estructura de detección, Snort se mantiene a la espera de recibir paquetes provenientes de la red para llevar a cabo el análisis de cada uno de ellos. En la figura 4.14 se muestra un diagrama de secuencia de ejecución de las funciones que están implicadas en esta fase de detección de tráfico, denominada fase *online*.

En este punto de la ejecución, en el que ya ha terminado la fase *offline*, el programa se encuentra en la función `SnortMain`, después de haber terminado de invocar a `SnortInit` y haber creado la estructura de detección. A continuación, se explicará el recorrido secuencial, paso a paso, continuando por la función `SnortMain`:

1. Antes de comenzar a recibir paquetes, Snort debe saber dónde ha de buscarlos. Hay varias fuentes en las que Snort puede obtener paquetes. Por defecto se obtienen de la red, pero también pueden ser leídos de un fichero pcap. Para decidir el modo de adquisición de paquetes, desde `SnortMain` se ejecuta la función `GetPacketSource`.

2. Una vez establecida la fuente a través de la cual se recibirán los paquetes, se inicia el módulo DAQ, también desde `SnortMain`, mediante las funciones `DAQ_Init` y `DAQ_New`. Se recuerda que el módulo DAQ es el encargado de recoger los paquetes de la red, o de cualquier otra fuente, y reenviarlos a Snort, pero esto no se llevará a cabo hasta el

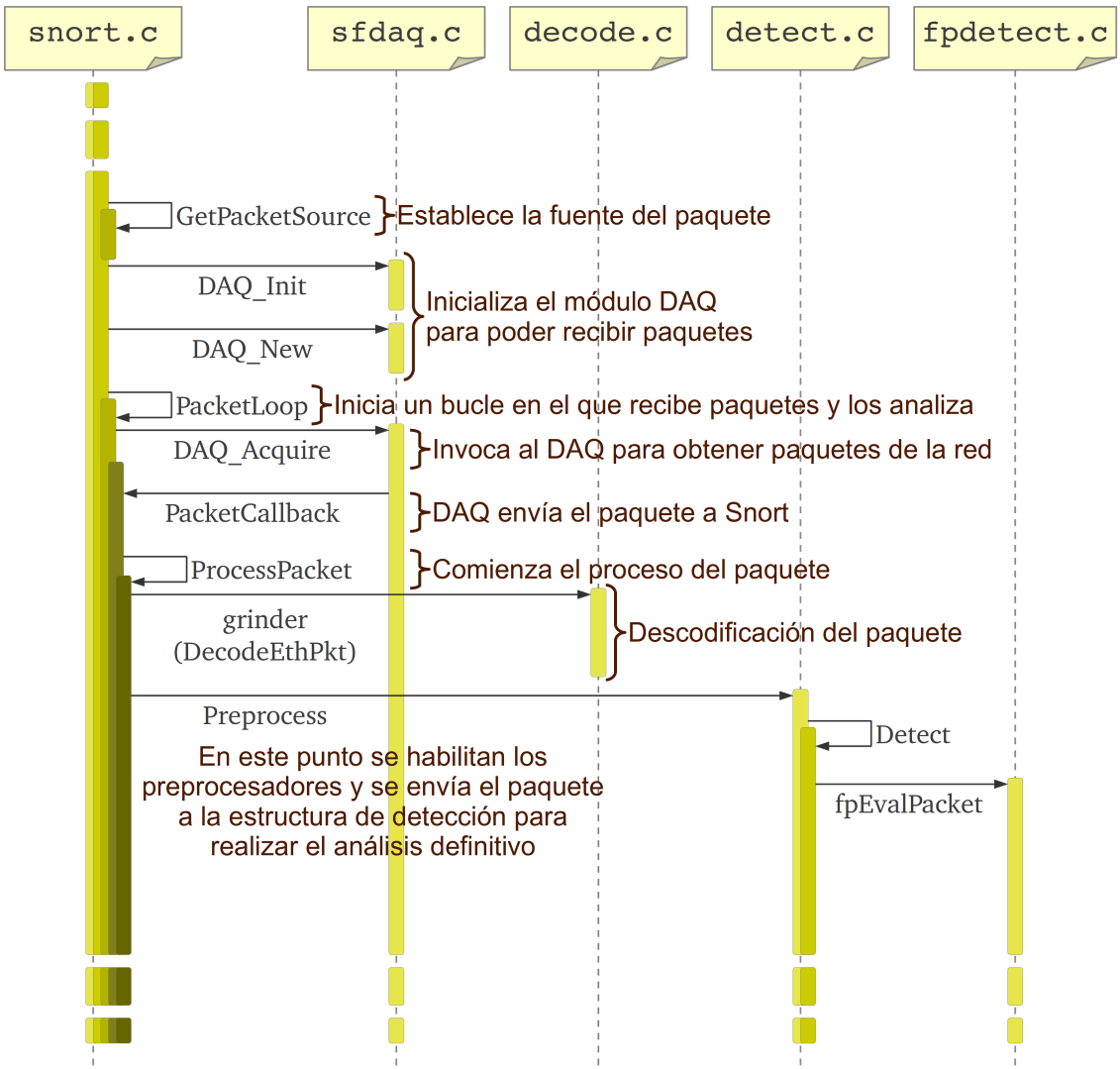

Figura 4.14: Diagrama de secuencia de la fase online de Snort.

paso 4.

3. A partir de ahora, `SnortMain` invoca a la función `PacketLoop`, que no devolverá el control a `SnortMain` hasta que no se termine la ejecución de la aplicación.

4. `PacketLoop` inicia un bucle "infinito" en el que siempre estará a la espera de recibir algún paquete. Saldrá del bucle al recibir una señal de terminación de programa o, en el caso de que la fuente sea un fichero pcap, cuando se termine de leer dicho fichero. Para ello ejecuta la función `DAQ_Acquire`, que devuelve un paquete mediante la llamada a la función `PacketCallback`, que se encuentra en el fichero snort.c.

5. Al recibir el paquete, la función `PacketCallback` realiza tareas de comprobación de señalización y evalúa contadores para las estadísticas. Por último invoca a la función `ProcessPacket` para llevar a cabo el análisis del paquete sobre la estructura de detección.

6. `ProcessPacket`, por su parte, ejecutará el descodificador de paquetes a través de la invocación del `grinder`, que previamente habrá almacenado la función que deberá lanzarse, por ejemplo `DecodeEthPkt`. Una vez descodificado el paquete, éste se envía al preprocesador mediante la llamada a la función `Preprocess`.

7. En la función `Preprocess` se distinguen dos tipos de módulos (o plug-ins) de preprocesador definidos en las estructuras `PreprocEvalFuncNode` y `PreprocReassemblyPktFuncNode`. La diferencia principal entre ellos es que los módulos del primer tipo sólo analizan el paquete, sin realizar ninguna modificación del mismo, mientras que los del segundo tipo sí lo modifican. En `Preprocess` primero se ejecutan todos los módulos que no modifican el paquete y después se invoca a la función `Detect`. Posteriormente, se ejecutan los módulos que sí modifican el paquete. Esta modificación no sobrescribe el paquete original, sino que se crea un duplicado del mismo para no perder su información. Después de la ejecución de cada módulo del segundo tipo se invoca a la función `Detect`. Es decir, se invocará tantas veces como módulos de preprocesador del segundo tipo haya. Para aclarar la explicación, en la figura 4.15 se muestra un esquema del funcionamiento de la función `Preprocess`.

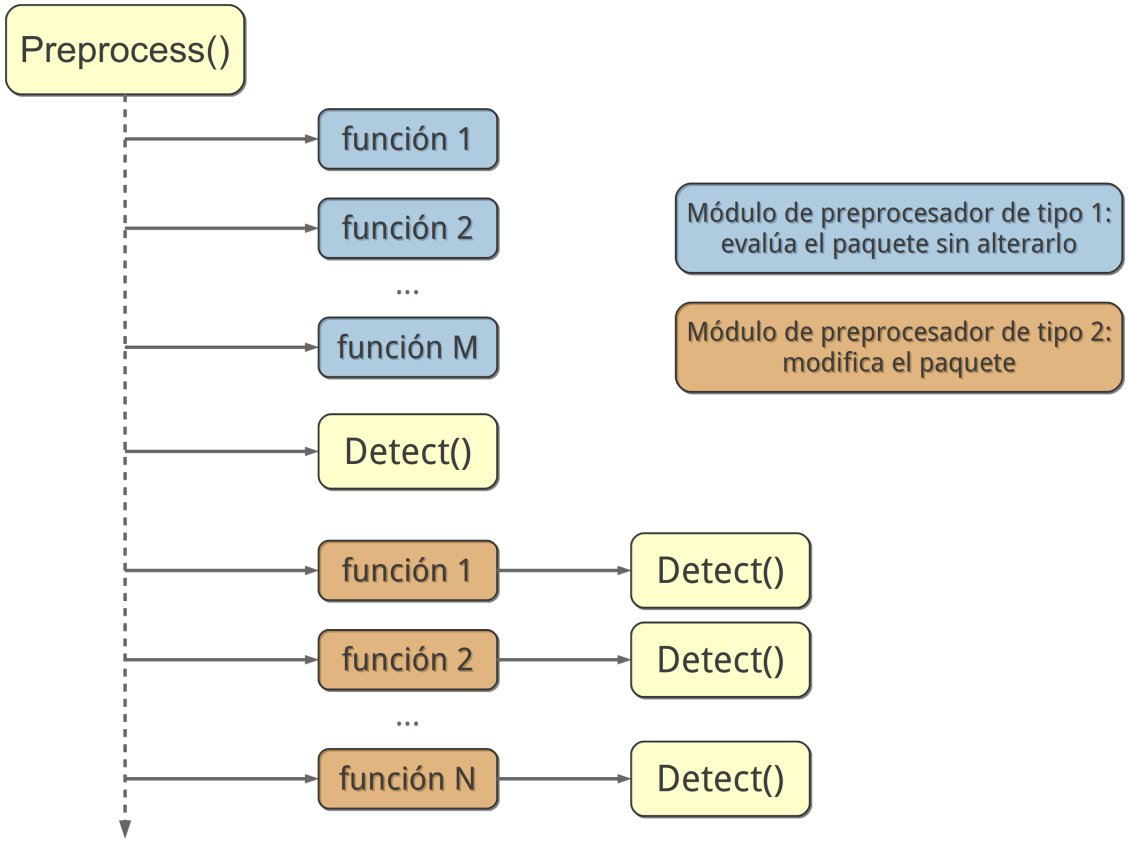

Figura 4.15: Esquema de la función Preprocess.

8. Posteriormente, la función `Detect` comprobará la integridad del paquete para verificar que pueda enviarse al motor de detección. Si el paquete no ha provocado ya una alerta en una inspección previa, entonces se ejecuta la función `fpEvalPacket`, que será la encargada de analizar el paquete mediante la llamada al motor de detección.

Del mismo modo que en la fase *offline*, en esta fase haremos hincapié en el punto en el que entra en juego el motor de detección. En este caso, explicaremos en detalle el camino que sigue la función `fpEvalPacket` hasta finalizar el análisis de un paquete determinado, puesto que esta parte del código de Snort es uno de los objetivos de estudio del proyecto. En la figura 4.16 se mostrará el camino a seguir de la aplicación desde que se invoca la función `fpEvalPacket` hasta que llega al motor de búsqueda multi-patrón (MPSE).

A continuación, se explicará cada función que forma parte del camino mostrado por la figura 4.16:

1. La función `fpEvalPacket` juega el rol de selector, es decir, al llegar el paquete a ella, ésta lo redirigirá a una de las cuatro funciones siguientes en base al protocolo del paquete en cuestión. En este punto de la ejecución el paquete se encontrará en el primer nivel de la lista enlazada de reglas, que es el nivel en el que se sitúan los protocolos. Lo siguiente será elegir hasta qué RTN se debe llegar.
2. Estas cuatro funciones son `fpEvalHeader{Tcp,Udp,Icmp,Ip}` y utilizan el protocolo y las direcciones IP y puertos de origen y destino del paquete para obtener la estructura `PORT_GROUP`. A través de esta estructura seleccionan el RTN al que pertenece el paquete para, posteriormente, enviar ambos, RTN y paquete, a la función `fpEvalHeaderSW`. Por tanto, en este punto, el paquete se encuentra en el segundo nivel de la lista enlazada 3D, es decir ya se sitúa sobre un RTN concreto.
3. Cuando se llega a la función `fpEvalHeaderSW`, ésta evalúa cada una de las reglas contenidas en cada OTN. Si alguna de las reglas incluye el campo `content` entonces, en su momento, se habría creado un DFA para realizar la búsqueda multi-patrón. En este caso,

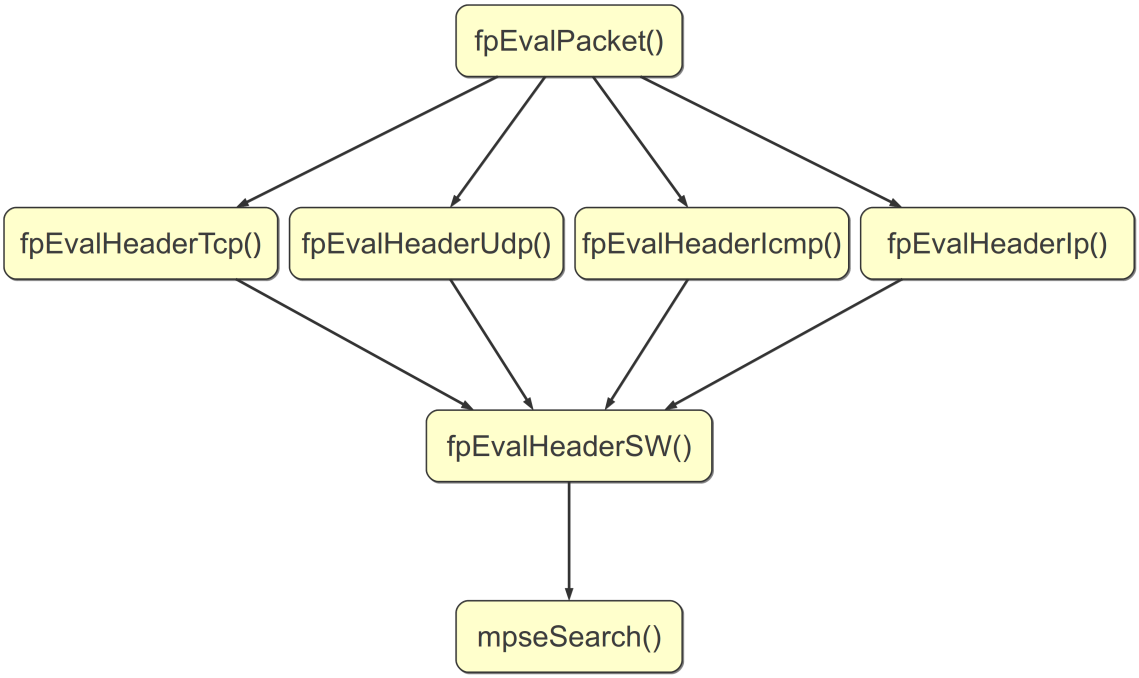

Figura 4.16: Esquema representativo del camino seguido desde la función fpEvalPacket hasta el motor de búsqueda multi-patrón (MPSE).

`fpEvalHeader` selecciona la estructura donde se almacena el DFA y la envía como parámetro en la llamada a la función `mpseSearch`. Esta última invoca al algoritmo de búsqueda multi-patrón especificado en el fichero de configuración de Snort.