Proyecto Final de Carrera

Sistemas de Prevención y Detección de Intrusos con Técnicas de Inteligencia Artificial

Alumno: Alexis M. Fredes Hadad

Director: Dr. Alejandro Javier García

Co-Director: Lic. Leonardo de Matteis

Ingeniería en Sistemas de Computación

Universidad Nacional del Sur – Año 2017

Tabla de contenido

Capítulo 1: Propuesta

1.1: Introducción

El propósito de este proyecto final es el estudio y análisis de las funcionalidades y clasificación de los diferentes tipos de sistemas de prevención y detección de intrusos. Asimismo, se realizará la instalación y experimentación del sistema específico y reciente de código abierto *Suricata*[1]. Dicha herramienta posee un motor de prevención y detección de intrusos que utiliza conjuntos de reglas desarrolladas externamente para monitorear el tráfico en la red y enviar alertas al administrador de sistemas cuando ocurren eventos sospechosos.

Otro de los objetivos es evaluar la efectividad y eficiencia de la herramienta a partir de un entorno de pruebas simulado.

Además, se estudiarán los nuevos paradigmas teóricos que incluyen la utilización de inteligencia artificial, redes neuronales, sistemas inmunológicos artificiales, etc. y su posible aplicación en los sistemas de detección de intrusos para poder identificar malware hasta ahora desconocido.

Por último, se evaluará la posibilidad de introducir dentro del código abierto alguna funcionalidad con técnicas de inteligencia artificial de las estudiadas en la etapa anterior.

1.2: Objetivos

- Estudio y análisis de las funcionalidades y clasificación de los diferentes tipos de sistemas de prevención y detección de intrusos.
- Instalación y experimentación de un sistema de detección de intrusos específico y reciente de código abierto en una máquina virtual.
- Evaluar la efectividad y eficiencia del sistema mencionado a partir de un entorno de pruebas simulado.
- Estudio de los nuevos paradigmas teóricos que incluyen la utilización de inteligencia artificial, sistemas inmunológicos artificiales, etc. y su aplicación en los sistemas de detección de intrusos.
- Posibilidad de introducir dentro del código abierto alguna funcionalidad con técnicas de inteligencia artificial de las estudiadas.

[1] https://suricata-ids.org/

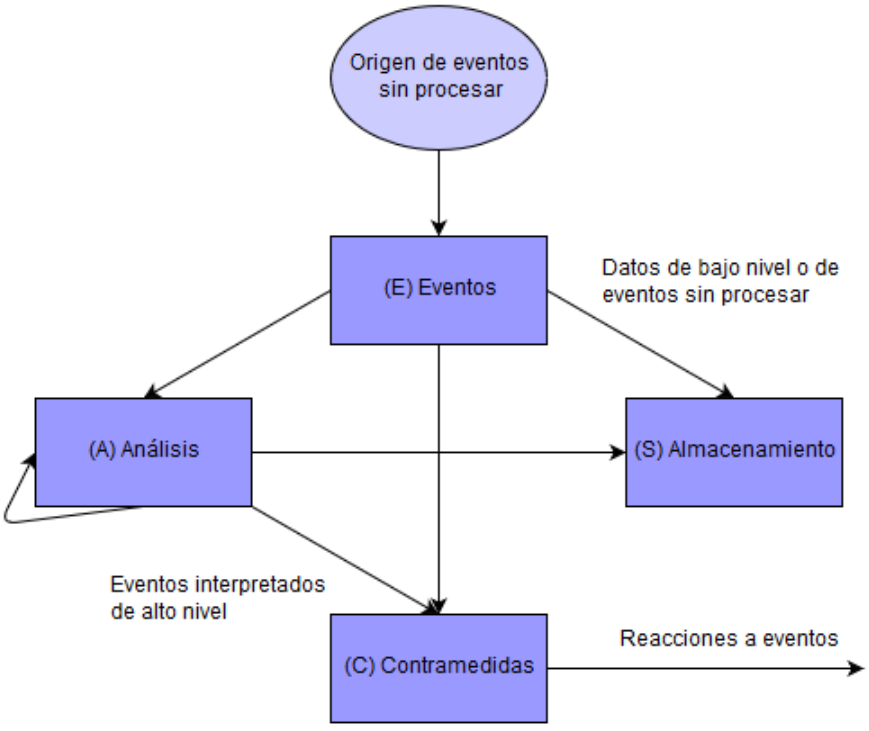
Capítulo 2: Sistemas de detección de intrusos

2.1: Definición

Un **sistema de detección de intrusos** (o **IDS** de sus siglas en inglés *Intrusion Detection System*) es una aplicación o dispositivo que monitorea una red o un sistema en busca de comportamientos malintencionados, violaciones a determinadas políticas y ataques. Estos últimos se manifiestan en términos de eventos, que pueden ser de diferente naturaleza y nivel de granularidad. Por ejemplo, los eventos pueden estar representados a través de paquetes de red, llamadas al sistema, registros de auditoria creados por herramientas del sistema operativo, o mensajes de registro de aplicaciones. El objetivo de un **IDS** es el de analizar flujos de eventos e identificar posibles ataques. Cuando posible un ataque es detectado, se genera una alerta que describe el tipo de ataque junto con las entidades involucradas (como pueden ser hosts, procesos, usuarios) que es enviada al administrador de seguridad del sistema.

Modelo de un IDS:

Como vemos en la imagen siguiente, un IDS recibe datos sin procesar a través de los sensores. Acto seguido, los almacena para analizarlos y una vez finalizado toma una acción de control.



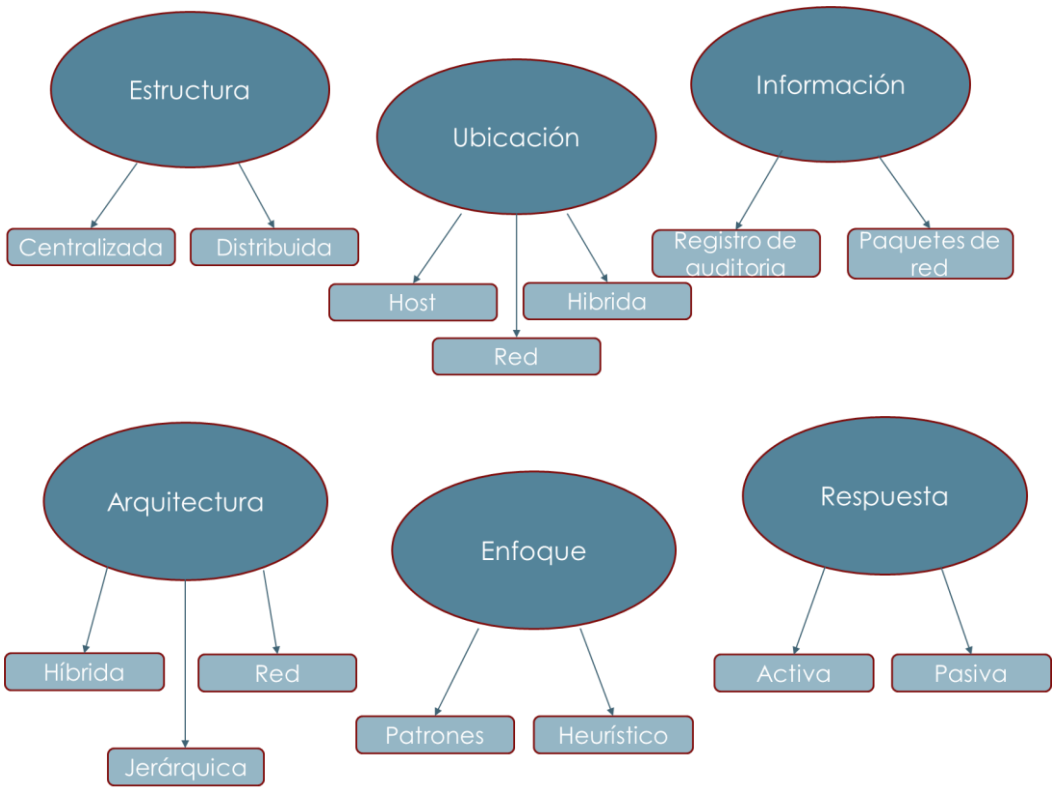
2.2: Funcionalidad

Un IDS realiza una gran cantidad de tareas entre las que se encuentran:

- Monitoreo de usuarios y de la actividad del sistema.
- Auditoria de la configuración del sistema con el fin de detectar vulnerabilidades y configuraciones erróneas.
- Evaluación de la integridad de sistemas críticos y archivos de datos.
- Reconocimiento de patrones de ataque conocidos en la actividad del sistema.
- Análisis estadísticos para identificar actividad anormal.
- Administración de los registros de auditoria para resaltar cuando se violan políticas o se detecta actividad normal.
- Corrección de errores de configuración del sistema.
- Instalación y operación de sistemas usados como cebo para recolectar información sobre intrusos (*honeypots*).

2.3: Clasificación

Los **IDS** pueden ser clasificados como se muestra en las imágenes siguientes:



Con el objetivo de poder realizar un estudio más preciso me centraré en los **IDS** clasificados de acuerdo a la ubicación donde la detección es realizada, y en el enfoque empleado.

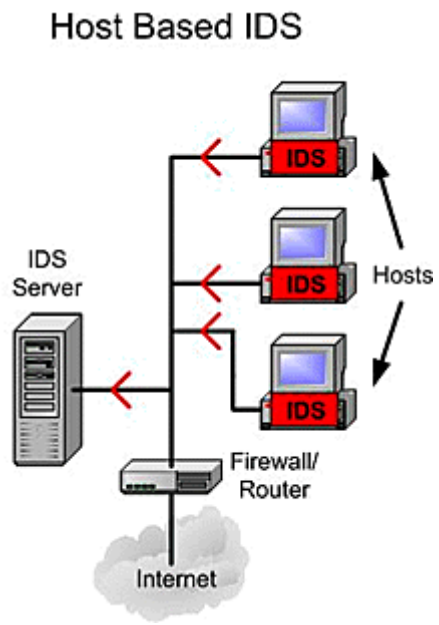
2.3.1: Ubicación

Host

Un **IDS** ubicado en el host (o **HIDS** por sus siglas en inglés, *Host Intrusion Detection System*) realiza la recolección y el análisis de información en una única computadora anfitriona para protegerla contra ataques. Para esto, monitorea en la capa de aplicación (es decir, accede a los servicios de las demás capas) los paquetes que entran y salen del dispositivo y lanza una alerta al usuario o administrador si se detecta actividad sospechosa. Asimismo, tiene la capacidad de realizar una captura de los archivos existentes del sistema para compararla con capturas previas con el fin de detectar si se eliminaron o modificaron archivos críticos.

El sistema operativo es el encargado de suministrar los datos al **IDS**, aprobando y denegando solicitudes de acceso a recursos sensibles, brindando registros de aplicaciones ejecutadas y demás información importante concerniente a la seguridad. El análisis de la información puede ser realizado en el mismo sistema o en uno independiente, en el último caso también se pueden correlacionar los datos con **HIDS** de otros hosts. Como desventaja se puede mencionar que como el **HIDS** es un proceso que se ejecuta en el sistema operativo del host está expuesto a ser detectado. Además, si un intruso desactiva el **HIDS**, el host será más vulnerable a posibles ataques.

En la siguiente imagen se aprecia la ubicación de varios **HIDS**.



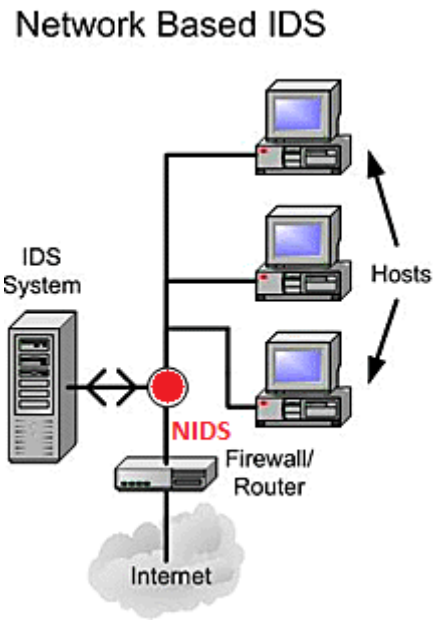
Red

Un **IDS** ubicado en la red (o **NIDS** por sus siglas en inglés, *Network Intrusion Detection System*) consiste generalmente de un dispositivo de red independiente ubicado en un lugar estratégico de la red que se encarga de analizar el tráfico que circula en la misma. Provee una segunda línea de defensa que detecta trafico anómalo en la capa física y de red una vez que se superó el firewall o alguna otra herramienta ubicada en la frontera de la red.

- Se identifican dos tipos:
  - Online: analizan la red en tiempo real y permiten realizar tareas de prevención.
  - Offline: analizan datos almacenados y esto les permite realizar tareas de corrección una vez que el daño ocurrió.

En ambos casos un **NIDS** recibe información de firewalls, sistemas operativos de las computadoras que pertenecen a la red, sensores que monitorean el volumen del tráfico y realizan balances de carga, y acciones que realiza el administrador en la red. El objetivo de un **NIDS** es el de proteger la totalidad de la red o un conjunto de recursos sensibles, como podría ser un grupo de servidores que almacenan información crítica, a través de la detección de anomalías tales como ataques de denegación de servicio. El software utilizado para la detección tiene la capacidad de analizar el contenido de los paquetes de red que circulan a través de la misma. Además, posee una librería de ataques conocidos que debe ser actualizada periódicamente para poder detectar, por ejemplo, acciones no habituales de un host (que fue comprometido) contra otro, y enviar una alerta al administrador del sistema.

En la imagen siguiente se aprecia una posible ubicación de un **NIDS**. En este caso está instalado en la misma subred donde se encuentra el firewall para así poder detectar si alguien está intentando atravesar el mismo. Idealmente, se debería realizar un escaneo de todo el tráfico entrante y saliente. Sin embargo, haciendo esto se crearía un cuello de botella que repercutiría en forma negativa el rendimiento de la red.



Un **NIDS** dispone de más herramientas para evitar ser detectado y comprometido en comparación con un **HIDS**, ya que puede actuar en un modo llamado “sigiloso” en el cual analiza la información de la red sin enviar nada a la misma para evitar ser detectado. Sin embargo, idealmente se requiere que ambas herramientas actúen en conjunto de modo que el **HIDS** pueda detectar una amenaza que logró sobrepasar al **NIDS**.

Otra funcionalidad de un **NIDS** es la de configurarse para disparar una alarma en una red independiente a la que está siendo monitoreada, de este modo la persona mal intencionada no se enterará que su ataque fue detectado.

Un punto a tener en cuenta es lo difícil de plasmar una configuración óptima para su ejecución ya que, de lo contrario, se obtendrán una gran cantidad de falsos positivos sumados a la gran cantidad de información que el administrador tendrá que procesar para poder advertir todos los ataques.

2.3.2: Enfoque

Patrones

Los **IDS** basados en patrones o firmas, (también llamados S*ignature IDS*), realizan la detección de ataques a partir de la búsqueda de patrones específicos como secuencias de bytes en el tráfico de red, o instrucciones maliciosas utilizadas por el malware.

Tienden a utilizar análisis estadístico que permite el uso de herramientas tanto para obtener muestras de indicadores clave (ej. número de transacciones y procesos activos), como para determinar si las mediciones recolectadas se ajustan a los patrones de ataque predeterminados.

Uno de los problemas que poseen este tipo de **IDS** son los patrones en los que basa el reconocimiento, ya que un atacante podría modificar un ataque básico de manera que su patrón no corresponda con el ubicado en la base de datos del **IDS**. Por ejemplo, el atacante podría intercambiar las letras en mayúscula por minúscula, o convertir un carácter como el del "espacio en blanco" a su equivalente "%20". El **IDS** debería trabajar con los datos en formato canónico para reconocer que dicho "%20" corresponde a un "espacio en blanco". También, el atacante podría tomar la estrategia de insertar paquetes falsos que el **IDS** reconocería, o mezclar el orden de los mismos para así lograr que el patrón sea incompatible con el almacenado. Cada una de estas variaciones se podrían incluir en la base de datos para ser detectadas, aunque una mayor cantidad de patrones implica trabajo adicional para el **IDS**, reduciendo su rendimiento.

Si bien estos IDS pueden detectar fácilmente ataques conocidos, les es imposible detectar nuevos ataques para los cuales no haya firma previamente instalada en la base de datos. Todo tipo de ataque comienza como un nuevo patrón en algún momento y el **IDS** no podrá detectar su existencia provocando un falso negativo. Estos tipos de ataques son llamados "*Zero Day Attacks*".

Como punto a destacar tenemos que estos tipos de **IDS** han demostrado tener una buena efectividad contra ciertos tipos de ataques de denegación de servicio (*DoS*) como *ping* y *echo-chargen* ya que poseen tipos de paquetes distinguibles. Por otro lado, para ataques como *teardrop*, donde se envían fragmentos de paquetes, la efectividad se reduce dado que se necesita recopilar información sobre todos o un número significativo de fragmentos. Y se vuelve prohibitivo teniendo en cuenta que la fragmentación de paquetes es una característica de la mayoría del tráfico, es decir que se tendrían que mantener todos los datos.

Heurística

Los **IDS** basados en heurística, (también llamados *Anomaly IDS*), surgieron para contrarrestar las limitaciones de los **IDS** basados en patrones teniendo en cuenta el rápido desarrollo del malware. En vez de buscar coincidencias, los **IDS** basados en heurísticas tienen como función la búsqueda de

comportamiento fuera de lo común como resultado de la comparativa con el comportamiento normal. Por ejemplo, un usuario podría comenzar siempre el día leyendo correos, escribiendo varios documentos a través del procesador de textos, y ocasionalmente resguardando archivos. Dichas acciones normales son utilizadas en la fase de entrenamiento para construir un perfil de comportamiento a través de *machine learning*. A partir de esto, se ejecuta la fase de testeo donde el comportamiento actual se compara con el perfil creado en la etapa anterior. Por lo tanto, si se da el caso en el que la persona trata de acceder o realizar alguna acción privilegiada, este nuevo comportamiento podría ser una pista de que alguna otra persona se está haciendo pasar por el usuario en cuestión. El esquema mencionado con anterioridad se encuentra limitado a la cantidad de información de comportamiento normal que el sistema ha recolectado.

El comportamiento anómalo puede ser detectado a través de técnicas de inteligencia artificial, redes neuronales, modelos matemáticos estrictos, data mining, sistemas inmunológicos artificiales, etc.

En el campo de la inteligencia artificial se utiliza una máquina de inferencia que analiza continuamente el sistema y emite una alerta cuando ocurre una combinación de factores que denotan un comportamiento malicioso. Es decir, se permitirán factores anómalos hasta superar un límite definido.

Una máquina de inferencia puede operar de dos maneras:

- La primera está basada en estados y su función es observar cómo el sistema va cambiando de estados o configuración a medida que pasa el tiempo. A partir de esto, tratará de detectar cuando el sistema cambia a un modo inseguro.
- La segunda forma de operación está basada en un modelo de actividad maliciosa conocida que dispara una alarma cuando la actividad actual se corresponde al modelo en cierto grado de completitud.

Aunque los *anomaly IDS* permiten la detección de ataques hasta ahora desconocidos, tienen como desventaja la gran cantidad de falsos positivos, ya que si se detecta una actividad legitima previamente desconocida será clasificada como anómala. Además, es realmente muy difícil implementar estos tipos de **IDS** en entornos dinámicos por el cambio constante en su comportamiento que poseen.

Comparativa con cortafuegos (o *firewalls*)

Si bien ambos pueden ser utilizados tanto en el ámbito de la seguridad en la red como en una computadora personal, un firewall realiza un monitoreo en la frontera para prevenir cualquier intrusión en la misma, es decir, actúa como una especie de barrera protectora que sirve de límite entre el sistema y el exterior. Sin embargo, este no será capaz de proteger al sistema cuando el ataque se genere dentro de la red como sucede en la mayoría de los casos o cuando se sobrepase su frontera de operación.

La mayor diferencia radica en que un **IDS** es un sistema reactivo que actúa corrigiendo en su totalidad o al menos disminuyendo el daño ocasionado por una intrusión que logró ingresar al sistema o red. Además, tiene una capacidad de respuesta contra ataques (malintencionados o no),

originados dentro de un sistema por parte de personas que pertenecen a la empresa u organización.

Todo esto es llevado a cabo a través de la inspección de la comunicación en la red, identificando heurísticas y patrones de ataques conocidos a partir del análisis de todo paquete de red. Posteriormente, realiza la comparación con su base de firmas para, de ser necesario, tomar las medidas pertinentes. Dichas medidas incluyen el aislamiento y la restricción de acceso del sospechoso a través de un sistema que finaliza conexiones llamado **Sistema de Prevención de Intrusos** (o **IPS** de sus siglas en inglés *Intrusion Prevention System*). Este último es otro tipo de control de acceso que actúa en la capa de red.

Para finalizar podemos afirmar que un IDS tiene una función diferente, aunque complementaria, a un *firewall*, ya que en un escenario ideal ambos deberían estas presentes.

Comparativa con antivirus

La diferencia entre ambas herramientas es cada vez menor ya que en el último tiempo los antivirus han ido sumando funcionalidad extra convirtiéndose en una verdadera suite de seguridad. Esto hace que para determinadas tareas exista una superposición de características.

Teniendo en cuenta que la principal finalidad que posee un antivirus es la de bloquear de forma activa el acceso a determinados archivos considerados como maliciosos, se puede afirmar que un IDS tiene un objetivo más amplio. Como la capacidad de identificar cambios en los sistemas de archivo, analizar archivos de registro, verificar los componentes de un sistema para detectar irregularidades, detectar potencial malware, entre otros.

Por otra parte, se puede afirmar que la mayoría de los antivirus que poseen versión gratuita actúan solamente como antivirus en la misma, mientras que la versión paga permite el acceso a la suite de seguridad completa. En este caso, gran parte de los IDS no limitan la funcionalidad en sus versiones gratuitas, aunque se debe pagar de manera adicional para acceder a una base de firmas más actual y completa.

Capítulo 3: Caso de estudio “Suricata”

3.1: Características

Suricata es una herramienta *open source*, rápida y robusta de detección de amenazas que implementa un Sistema de Detección y Prevención de Intrusos ubicado en la Red (o **NIDPS** por sus siglas en inglés, *Network Intrusion Detection and Prevention System*). Es desarrollado por una organización sin fines de lucro llamada Fundación Abierta de Seguridad de la Información (o **OISF** por sus siglas en inglés, *Open Information Security Foundation*) cuyo objetivo es el de brindar nuevas ideas e innovaciones tecnológicas a la industria de detección de intrusos.

La principal ventaja de Suricata frente a las demás herramientas es la de poseer una arquitectura *multi-threaded* que permite un alto rendimiento en sistemas de múltiples núcleos y/o procesadores. Esto permite ofrecer un análisis de tráfico de red más veloz y eficiente, y puede dividir la carga de trabajo de los sistemas de detección y prevención de intrusos en base a donde se requiera realizar el procesamiento. Además de la aceleración por hardware, el motor fue desarrollado para aprovechar los mayores beneficios que ofrecen los últimos procesadores de varios núcleos. También, ofrece la posibilidad aprovechar la arquitectura de cálculo paralelo CUDA de Nvidia para aprovechar la gran potencia de la GPU (unidad de procesamiento gráfico).

Suricata fue desarrollado poniendo el foco en la facilidad de implementación, acompañado por un manual de usuario y un mail list donde se puede recibir soporte en caso de necesidad. El motor que utiliza está escrito en el lenguaje de programación C y fue diseñado para ser escalable.

Es importante destacar que se utilizará la versión 3.2.1, la cual es la correspondiente al 15 de febrero de 2017.

Entre sus principales características se encuentran:

- Prevención y detección de intrusos en tiempo real (IPS en capa de red, IDS en capa de aplicación).
- Monitoreo de seguridad en redes (NSM)[2].

[2] NSM (Network Security Monitoring) es la recolección, análisis y escalamiento de indicaciones y advertencias para detectar y responder a intrusiones.

- Inspecciones de tráfico para amenazas conocidas a través de un lenguaje extendido de Snort lo que asegura su compatibilidad.
- Posesión de un motor open source. Esto hace que se trabaje como una comunidad al momento de capturar las características de las amenazas que emergen.
- Soporte de reputación de IPs a través de la incorporación de firmas a su motor, lo que le permite detectar el tráfico de fuentes maliciosas conocidas.
- Detección mediante scripts basados en el lenguaje Lua (multiparadigma[3], interpretado[4] e imperativo[5]).
- Datos de salida en JSON (JavaScript Object Notation) para facilitar el post-procesamiento.
- Capacidad de realizar extracción de archivos.
- Captura y procesamiento offline de paquetes (PCAP).
- Soporte IPv6.
- Aceleración nativa por hardware.
- Escalable a través de multi-threading. Permite que el motor pueda aprovechar las ventajas de las arquitecturas de múltiples núcleos y/o procesadores de los sistemas actuales.
- Detección automática de protocolos. A través de un pre-procesamiento se identifica el protocolo que utiliza un determinado flujo de red y aplica las reglas apropiadas sin importar el número de puerto.

¿Por qué Suricata es tan exitoso en el monitoreo de ataques sofisticados?

Suricata tiene la capacidad de utilizar conjuntos de reglas desarrolladas externamente para monitorear el tráfico de red y así, emitir alertas al administrador del sistema cuando ocurren eventos sospechosos. Además, usa un motor de *sniffing* para analizar todo el tráfico entrante y saliente. Esto sumado a la capacidad *multi-threading* permite que el sniffer pueda combinar rápidamente más reglas y aplicar potencia computacional adicional al proceso de seguridad.

También, Suricata fue diseñado para ser compatible con los componentes de red existentes a través de la funcionalidad de salida que provee *Unified2* y opciones de biblioteca conectables para aceptar llamadas de otras aplicaciones. En este sentido, Suricata es compatible con el conjunto de reglas de *Snort*. Asimismo, el motor posee un normalizador y analizador HTTP que brinda un procesamiento de los flujos HTTP más avanzado, permitiendo comprender el tráfico en la 7ta capa del modelo OSI.

[3] Soporta más de un paradigma de programación. Permite a los programadores utilizar el mejor paradigma para cada trabajo, admitiendo que ninguno resuelve todos los problemas de la forma más fácil y eficiente posible.
[4] La mayoría de sus implementaciones ejecutan las instrucciones directamente, sin una previa compilación del programa a instrucciones en lenguaje máquina.
[5] Describe la programación en términos del estado del programa y sentencias que cambian dicho estado.

3.2: Instalación

Para que la instalación de Suricata pueda ser llevada a cabo, primero se deben realizar una serie de pasos:

- Software de virtualización
  Instalación de un software de virtualización llamado VMware para garantizar la abstracción del software con respecto al hardware y así posibilitar una mayor flexibilidad a la hora de migrar el sistema de una computadora a otra.

- Sistema operativo
  Luego de realizar pruebas sobre diferentes distribuciones Linux para identificar la adecuada se detectó que **OISF** mantiene un **A**rchivo de **P**aquete **P**ersonal[6], o **APP** (en inglés *Personal Package Archive*, o **PPA**) para Ubuntu y, por ende, posee en sus repositorios la versión estable más reciente de Suricata. Esto permite evitar complicaciones adicionales a la hora de instalar el **IDS**. Además, Ubuntu tiene una gran comunidad de desarrolladores y soporte. La versión a utilizar será la 16.04 LTS (*Long Term Support*) llamada *Xenial Xerus*.

Luego de satisfacer los requerimientos mencionados con anterioridad ejecutamos el comando para actualizar la lista de paquetes disponibles de Ubuntu.

```
ale@ubuntu:~$ sudo apt-get update
```

Y descargamos la última versión de dichos paquetes.

```
ale@ubuntu:~$ sudo apt-get upgrade
```

La instalación de Suricata puede hacerse de varias maneras dependiendo el control que queramos poseer en la misma:

- A través de los repositorios, desde donde se instala la versión estable más reciente de la siguiente manera:
  Se procede a agregar los repositorios de OISF a Ubuntu

  ```
  ale@ubuntu:~$ sudo add-apt-repository ppa:oisf/suricata-stable
  ```

  Luego se actualizan los repositorios como se hizo en el primer paso. La diferencia es que ahora se incluirá el repositorio agregado en el paso previo.

[6] Es un repositorio de software especial en el que se suben paquetes fuente para ser construidos y publicados como un repositorio en el sistema de gestión de paquetes (APT).

Por último, se instala Suricata con las funcionalidades de IDS e IPS activas.

```
ale@ubuntu:~$ sudo apt-get install suricata
```

Se comprueba la versión instalada.

```
ale@ubuntu:~$ suricata -V
This is Suricata version 3.2.1 RELEASE
ale@ubuntu:~$
```

Es importante mencionar que de esta manera Hyperscan se instalará solo a partir de Ubuntu 17.04. De lo contrario se debe realizar la instalación a través de los archivos de distribución de origen.

- A través de los archivos de distribución de origen de la siguiente manera:

  Es importante mencionar que, si se desea añadir la biblioteca *Hyperscan* a Suricata, se debe instalar previamente dicha biblioteca aquí, para luego proseguir con la corriente instalación.

  Siguiendo con la instalación recomendada de Suricata se deben añadir las siguientes dependencias:

  - Por el lado de las librerías y sus encabezados de desarrollo se deben instalar:
    - *libpcre*; la cual es un conjunto de instrucciones que implementa pattern matching de expresiones regulares a través de la sintaxis y semántica de Perl 5.

      ```
      ale@ubuntu:~$ sudo apt-get install libpcre3 libpcre3-dbg libpcre3-dev
      ```

    - *libcap-ng*; librería cuyo objetivo es el de facilitar la programación de capacidades *posix*.

      ```
      ale@ubuntu:~$ sudo apt-get install libcap-ng-dev libcap-ng0
      ```

    - *libpcap*; es la implementación de una aplicación de programación para captura de paquetes en sistemas basados en Unix.

      ```
      ale@ubuntu:~$ sudo apt-get install libcap-dev
      ```

    - *libyaml*; es la librería de YAML escrita en C. YAML es un formato de almacenamiento de datos diseñado para facilitar su lectura y análisis. Es más detallado que JSON.

      ```
      ale@ubuntu:~$ sudo apt-get install libyaml-0-2 libyaml-dev
      ```

    - *libmagic*; reconoce números que poseen formato *magic*. Estos números están compuestos por caracteres alfanuméricos que, de manera codificada, identifican un archivo.

      ```
      ale@ubuntu:~$ sudo apt-get install libmagic-dev
      ```

    - *libjansson*; librería C para la codificación, decodificación y manipulación de datos en formato JSON.

      ```
      ale@ubuntu:~$ sudo apt-get install libjansson-dev
      ```

    - *zlib1g*; es una biblioteca open source de compresión de datos.

      ```
      ale@ubuntu:~$ sudo apt-get install zlib1g zlib1g-dev
      ```

- *libnss3*; conjunto de bibliotecas diseñadas para permitir el desarrollo multiplataforma de aplicaciones cliente y servidor con seguridad habilitada.

```
ale@ubuntu:~$ sudo apt-get install libnss3-dev
```

- *libgeoip*; librería que le permite al usuario encontrar el país de donde proviene una dirección IP.

```
ale@ubuntu:~$ sudo apt-get install libgeoip-dev
```

- *liblua5.1*; lenguaje de programación liviano y potente que permite extender aplicaciones a través del agregado de funciones.

```
ale@ubuntu:~$ sudo apt-get install liblua5.1-dev
```

- *libhiredis*; librería minimalista C utilizada en la base de datos Redis.

```
ale@ubuntu:~$ sudo apt-get install libhiredis-dev
```

- *libnet1*; es una librería que provee un framework para la construcción e inyección de paquetes. Es necesario si queremos utilizar la acción *Reject* ya que en este caso el sistema actúa como un IPS.

```
ale@ubuntu:~$ sudo apt-get install libnet1-dev
```

- Por el lado de las herramientas, se deben añadir:
  - *pkg-config*; provee una interfaz unificada para llamar bibliotecas instaladas cuando se está compilando un programa a partir del código fuente.

```
ale@ubuntu:~$ sudo apt-get install pkg-config
```

  - *gcc*; conjunto de compiladores de varios lenguajes de programación.

```
ale@ubuntu:~$ sudo apt-get install gcc
```

  - *make*; herramienta que controla la generación de ejecutables de un programa a partir de sus archivos fuente.

```
ale@ubuntu:~$ sudo apt-get install make
```

  - *build-essential*; Este paquete contiene una lista informativa de los paquetes considerados esenciales para la creación de paquetes Debian/ GNU Linux.

```
ale@ubuntu:~$ sudo apt-get install build-essential
```

Si además se desea utilizar Suricata como una herramienta de prevención de intrusos se deben añadir los siguientes extras relacionados con *iptables/nftables*:

- *libnetfilter-queue* y *libnetfilter-log*; librerías que se encargan de proveer una API a los paquetes que fueron encolados por el filtrador de paquetes del kernel.

```
ale@ubuntu:~$ sudo apt-get install libnetfilter-queue-dev libnetfilter-queue1
```

```
ale@ubuntu:~$ sudo apt-get install libnetfilter-log-dev libnetfilter-log1
```

- *libnfnetlink*; librería de bajo nivel que provee una infraestructura de mensajería entre subsistemas ubicados en el kernel y en el espacio del usuario.

```
ale@ubuntu:~$ sudo apt-get install libnfnetlink-dev libnfnetlink0
```

Luego descargamos Suricata desde su página oficial y descomprimimos el archivo.

```
ale@ubuntu:~/Downloads$ tar xzvf suricata-3.2.1.tar.gz
```

Accedemos a la carpeta que posee los archivos de Suricata.

```
ale@ubuntu:~/Downloads$ cd suricata-3.2.1/
```

Configuramos Suricata (solo IDS):

```
ale@ubuntu:~/Downloads/suricata-3.2.1$ ./configure --prefix=/usr --sysconfdir=/etc --localstatedir=/var
```

Configuramos Suricata (IDS e IPS):

```
ale@ubuntu:~/Downloads/suricata-3.2.1$ ./configure --enable-nfqueue --prefix=/usr --sysconfdir=/etc --localstatedir=/var
```

Como resultado podemos visualizar las funcionalidades extra que estarán listas para ser utilizadas en Suricata, como por ejemplo el soporte a *Hyperscan* y la función NFQueue si es que se configuro el sistema como IPS.

```
Suricata Configuration:
  AF_PACKET support:                       yes
  PF_RING support:                         no
  NFQueue support:                         yes
  NFLOG support:                           no
  IPFW support:                            no
  Netmap support:                          no
  DAG enabled:                             no
  Napatech enabled:                        no

  Unix socket enabled:                     yes
  Detection enabled:                       yes

  Libmagic support:                        yes
  libnss support:                          yes
  libnspr support:                         yes
  libjansson support:                      yes
  hiredis support:                         no
  Prelude support:                         no
  PCRE jit:                                yes
  LUA support:                             no
  libluajit:                               no
  libgeoip:                                no
  Non-bundled htp:                         no
  Old barnyard2 support:                   no
  CUDA enabled:                            no
  Hyperscan support:                       yes
  Libnet support:                          yes

  Suricatasc install:                      yes

  Profiling enabled:                       no
  Profiling locks enabled:                 no

Development settings:
  Coccinelle / spatch:                     no
  Unit tests enabled:                      no
  Debug output enabled:                    no
  Debug validation enabled:                no
```

Por último, ejecutamos los comandos *make* y *sudo make install-conf* para realizar la instalación de Suricata. El *conf* se encargará de crear y configurar todos las carpetas necesarias y el archivo *suricata.yaml* una vez realizada la instalación.

Descripción breve de uso.

```
ale@ubuntu:~$ suricata -h
Suricata 3.2.1
USAGE: suricata [OPTIONS] [BPF FILTER]

        -c <path>                           : path to configuration file
        -T                                  : test configuration file (use with -c)
        -i <dev or ip>                      : run in pcap live mode
        -F <bpf filter file>                : bpf filter file
        -r <path>                           : run in pcap file/offline mode
        -q <qid>                            : run in inline nfqueue mode
        -s <path>                           : path to signature file loaded in addition to suricata.yaml settings (optional)
        -S <path>                           : path to signature file loaded exclusively (optional)
        -l <dir>                            : default log directory
        -D                                  : run as daemon
        -k [all|none]                       : force checksum check (all) or disabled it (none)
        -V                                  : display Suricata version
        -v[v]                               : increase default Suricata verbosity
        --list-app-layer-protos             : list supported app layer protocols
        --list-keywords[=all|csv|<kword>]   : list keywords implemented by the engine
        --list-runmodes                     : list supported runmodes
        --runmode <runmode_id>              : specific runmode modification the engine should run.  The argument
                                              supplied should be the id for the runmode obtained by running
                                              --list-runmodes
        --engine-analysis                   : print reports on analysis of different sections in the engine and exit.
                                              Please have a look at the conf parameter engine-analysis on what reports
                                              can be printed
        --pidfile <file>                    : write pid to this file
        --init-errors-fatal                 : enable fatal failure on signature init error
        --disable-detection                 : disable detection engine
        --dump-config                       : show the running configuration
        --build-info                        : display build information
        --pcap[=<dev>]                      : run in pcap mode, no value select interfaces from suricata.yaml
        --pcap-buffer-size                  : size of the pcap buffer value from 0 - 2147483647
        --af-packet[=<dev>]                 : run in af-packet mode, no value select interfaces from suricata.yaml
        --simulate-ips                      : force engine into IPS mode. Useful for QA
        --user <user>                       : run suricata as this user after init
        --group <group>                     : run suricata as this group after init
        --erf-in <path>                     : process an ERF file
        --unix-socket[=<file>]              : use unix socket to control suricata work
        --set name=value                    : set a configuration value


To run the engine with default configuration on interface eth0 with signature file "signatures.rules", run the command as:

suricata -c suricata.yaml -s signatures.rules -i eth0
```

Ejecución de Suricata:

Puede suceder que ocurra el siguiente error al intentar correr Suricata por primera vez:

```
ale@ubuntu:/etc/suricata$ sudo LD_PRELOAD="/usr/lib/libtcmalloc_minimal.so.4" suricata -c suricata.yaml -i ens33
17/7/2017 -- 19:14:45 - <Notice> - This is Suricata version 3.2.1 RELEASE
17/7/2017 -- 19:15:05 - <Error> - [ERRCODE: SC_ERR_INITIALIZATION(45)] - Cannot create socket directory /usr/local/var/run/suricata/: No such file or directory
17/7/2017 -- 19:15:05 - <Warning> - [ERRCODE: SC_ERR_INITIALIZATION(45)] - Unable to create unix command socket
```

Esto se debe a que Suricata necesita crear un socket para su comunicación y no existe la ubicación adecuada para crear el mismo. Para solucionarlo, nos dirigimos al directorio */usr/local* y creamos las carpetas *var*, *run*, *suricata* con el comando *sudo mkdir* de modo que la ruta nos quede de la forma */usr/local/var/run/suricata*.

En una placa de red Wireless:

```
ale@ubuntu:~$ sudo suricata -c /etc/suricata/suricata.yaml -i wlx002191960e38
```

En una placa de red Ethernet:

```
ale@ubuntu:~$ sudo suricata -c /etc/suricata/suricata.yaml -i ens33
```

3.3: Reglas

Las firmas juegan un rol muy importante en Suricata. En muchas ocasiones se utilizan conjuntos de reglas existentes. Las más utilizadas son *Emerging Threats*, *Emerging Threats Pro* y *Sourcefire's VRT*. Una manera de descargar e instalar firmas de forma automática es a través de la herramienta Oinkmaster. Este proceso será descripto en la sección Gestión de reglas.

Una firma/regla se compone por tres elementos: una acción, un encabezado y sus opciones.

```
drop tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"ET
TROJAN Likely Bot
Nick in IRC (USA +..)"; flow:established,to_server;
flowbits:isset,is_proto_irc; content:"NICK "; pcre:"/NICK
.*USA.*[0-9]{3,}/i"; classtype:trojan-activity;
reference:url,doc.emergingthreats.net/2008124;
reference:url,www.emergingthreats.net/cgi-
bin/cvsweb.cgi/sigs/VIRUS/TROJAN_IRC_Bots;
sid:2008124; rev:2;)
```

Orden de acción

Como vimos con anterioridad todas las firmas/reglas poseen distintos componentes. Una de ellas es la acción, la cual determina que es lo que sucede cuando ocurre una coincidencia con la firma (match). Se identifican cuatro tipos de acciones:

1. **Aprobar (*pass*):** si una firma empareja y contiene esta acción, Suricata detiene el escaneo del paquete y omite las reglas restantes para el paquete en cuestión.
2. **Soltar (*drop*):** esto solo se aplica en el modo IPS/en línea. Si el programa encuentra una firma que coincide y contiene dicha acción, el paquete se detiene de forma inmediata, es decir que no será enviado más. El inconveniente que surge es que el receptor del paquete no recibirá el paquete en cuestión ni un mensaje sobre lo que ha pasado, por lo tanto, ocurre un *time-out* (con *TCP*). Suricata generará una alerta para este paquete.
3. **Rechazar (*reject*):** consta de un rechazo activo del paquete, por lo que el emisor y el receptor recibirán un rechazo de paquete. Hay dos tipos de paquetes de rechazo entre los cuales se seleccionará uno de forma automática. Si el paquete ofensivo utiliza TCP, será un paquete de *reset*, de lo contrario será un paquete de error ICMP. Además, Suricata generará una alerta. Cuando el programa se encuentra en el modo IPS/en línea el paquete ofensivo será soltado como sucede con la acción anterior.
4. **Alertar (*alert*):** cuando ocurre el emparejamiento, el paquete será tratado como si no fuera una amenaza con la excepción que Suricata generará una alerta para el mismo que solo será advertida por el administrador de seguridad.

IPS/en línea puede realizar el bloqueo de un paquete de dos maneras distintas. Una es soltando el paquete (*drop*) y la otra es rechazándolo.

Las reglas serán cargadas en el orden en que aparecen en los archivos, pero serán procesadas en un orden distinto. Las firmas poseen prioridades diferentes lo que permite que el escaneo empiece por las de mayor prioridad. El orden de prioridad puede ser modificado. El orden de acción predeterminado es: aprobar, soltar, rechazar y alertar.

```
action-order:
 - pass
 - drop
 - reject
 - alert
```

Lo que significa que una regla de aprobar será considerada antes que una de soltar, una de soltar antes que una de rechazar, y así sucesivamente. Ubicación: línea 981 en suricata.yaml.

Protocolo

Esta palabra clave es la encargada de especificarle a Suricata que protocolo es el utilizado. Se ubica al principio del encabezado y se puede elegir entre cuatro configuraciones: TCP, UDP, ICMP e IP, este último es el utilizado para “todos” o “cualquiera”. Suricata añade otros protocolos que se ejecutan en la capa de aplicación como HTTP, FTP, TLS (incluye SSL), SMB y DNS (desde versión 2). Por ejemplo, si una firma posee el protocolo HTTP, Suricata se asegura que el emparejamiento exista solo dentro de tráfico HTTP.

Direcciones origen y destino

En origen se pueden asignar solo direcciones IPv4, IPv6, o una combinación de las mismas. Además, se pueden crear variables como *HOME_NET* (imagen inferior) para agrupar un conjunto de direcciones IP y luego ser usadas en reglas. Dentro de cada regla existe la posibilidad de establecer para que dirección IP debe comprobarse la regla y para qué dirección IP no debería. De este modo, solo se utilizarán las reglas que son relevantes. Para prevenir que el usuario tenga que establecer una dirección IP relevante en cada regla se puede utilizar una opción para definir la dirección para un grupo de reglas.

En la imagen inferior *$HOME_NET* corresponde a la fuente mientras que *$EXTERNAL_NET* al destino.

```
drop tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"ET
TROJAN Likely Bot
Nick in IRC (USA +..)"; flow:established,to_server;
flowbits:isset,is_proto_irc; content:"NICK "; pcre:"/NICK
.*USA.*[0-9]{3,}/i"; classtype:trojan-activity;
reference:url,doc.emergingthreats.net/2008124;
reference:url,www.emergingthreats.net/cgi-
bin/cvsweb.cgi/sigs/VIRUS/TROJAN_IRC_Bots;
sid:2008124; rev:2;)
```

Puertos origen y destino

El tráfico entra y sale a través de puertos, cada uno de estos posee un número que lo distingue. Normalmente el puerto origen se configura para que se acepte cualquiera (*any* en la imagen superior), aunque esto será afectado por el protocolo. Además, el sistema operativo es el encargado de realizar la elección aleatoria del puerto origen. También es posible realizar un filtrado de puertos a través de comandos especiales como, por ejemplo:

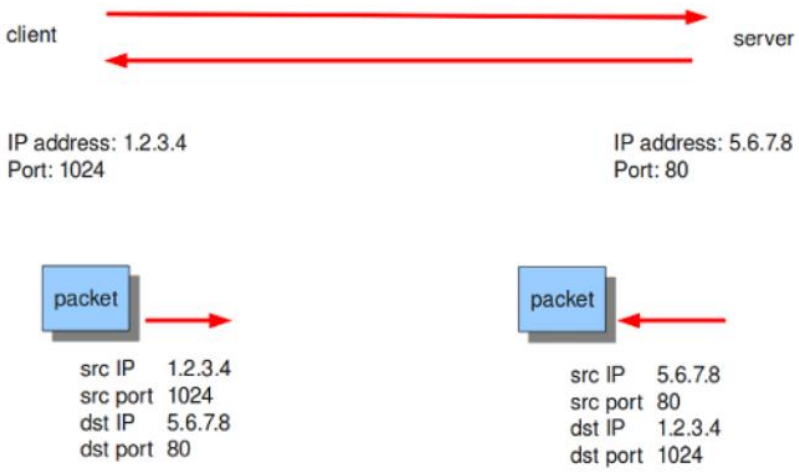
```
[80, 81, 82]     (port 80, 81 and 82)
[80: 82]         (Range from 80 till 82)
[1024: ]         (From 1024 till the highest port-number)
!80              (Every port but 80)
[80:100,!99]     (Range from 80 till 100 but 99 excluded)
[1:80,![2,4]]
[....[.....]]
```

Sentido

El sentido nos posibilita especificar en qué sentido se realizará el emparejamiento de la firma.

```
source -> destination
source <> destination  (both directions)
```

Esto quiere decir que podrán coincidir solo los paquetes que vayan en la misma dirección. A modo de ejemplo, si especificamos `alert tcp 1.2.3.4 1024 -> 5.6.7.8 80` solo habrá emparejamiento en el paquete que va desde el cliente al servidor.



Meta-configuraciones

Inciden en la manera en la que Suricata informa los eventos y no tienen ningún efecto en la inspección que realiza este IDS. Se compone por un conjunto de palabras clave:

1. Mensajes (*message*): *msg* brinda más información sobre la firma y la posible alerta. La primera parte se escribe en mayúscula por convención y muestra la clase de firma. Ejemplo: `msg:"ATTACK-RESPONSES 403 Forbidden";`
2. Identificación de firma (*signature id*): *sid* otorga un número de identificación numérico a cada firma. Ejemplo, `sid:123;`
3. Revisión (*rev*): acompaña a la palabra clave *sid*. Representa la versión de la firma, es decir que si una firma es modificada se incrementara el número de revisión. Por convención *sid* va ubicado antes que *rev* y ambos son los últimos de todas las palabras claves. Ejemplo, `rev:2;`
4. Identificación de grupo (*group id*): su función es la de brindar otra identificación (*gid*) a un grupo de firmas. Su funcionamiento es similar a *sid*. Un ejemplo extraído del archivo *fast.log* seria:

   ```
   10/15/09-03:30:10.219671 [**] [1:2008124:2] ET TROJAN
   Likely Bot Nick
   in IRC (USA +..) [**] [Classification: A Network Trojan was
   Detected]
   [Priority: 3] {TCP} 192.168.1.42:1028 -> 72.184.196.31:6667
   ```

   En la parte [1:2008124:2], 1 es el *gid* (2008124 es el *sid* y 2 es el *rev*).
5. Clasificación por tipo (*classtype*): brinda información sobre la clasificación de reglas y alertas. Una o más reglas se agrupan dentro de una clase de la que heredarán la prioridad. Consiste de un nombre breve seguido de un nombre completo y una prioridad. Por convención se lo ubica antes que *sid* y *rev*. Ejemplo: en archivo `classification.config: trojan-activity,A Network Trojan was Detected, 1.`
6. Referencia (*reference*): se encarga de ubicar la información sobre la firma y sobre el problema a la que la firma trata de dirigirse. Esta palabra clave puede aparecer varias veces por cada firma. Ejemplo: `reference: url, www.info.nl.`
7. Prioridad (*priority*): su función es la de otorgar prioridad a una firma a través de un valor numérico obligatorio que puede ir desde 1 a 255. La prioridad asciende a medida que el valor es más pequeño y hace que la firma sea evaluada primero. Normalmente las firmas

tienen una prioridad heredada por la clase a la que pertenecen (*classtype*) que puede ser anulada por la siguiente palabra clave. Ejemplo: `priority:1;`

8. Metadatos (*metadata*): provoca que Suricata ignore las palabras que están detrás. Ejemplo: `metadata…;`

Palabras clave del encabezado

- Pertenecientes al IP:
  - TTL (*time to live*): es utilizado para chequear el valor correspondiente al número máximo de routers que puede atravesar un paquete antes de llegar a destino, evitando que se produzcan ciclos infinitos. Si su valor es 0 entonces el paquete tiene que ser destruido. Ejemplo, `ttl:10;`
  - Ipopts (*ip option set*): su función es la de chequear si alguna opción IP fue establecida. Se puede especificar una sola opción por regla y debe estar al comienzo de la misma. Su formato es: `ipopts: <name>` y las opciones disponibles son:

IP-option	Description
rr	Record Route
eol	End of List
nop	No Op
ts	Time Stamp
sec	IP Security
esec	IP Extended Security
lsrr	Loose Source Routing
ssrr	Strict Source Routing
satid	Stream Identifier
any	any IP options are set

  - Sameip: realiza la comprobación si la dirección IP fuente es la misma que la de destino en un paquete. Su formato es: `sameip;`
  - Ip_proto: se utiliza para emparejar el protocolo IP con el encabezado del paquete. Se puede utilizar el nombre o el número de protocolo. Algunos de los protocolos son:

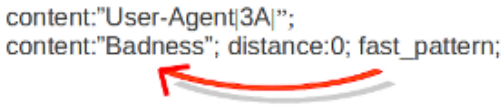
```
  1     ICMP        Internet Control Message
  6     TCP         Transmission Control Protocol
 17     UDP         User Datagram
 47     GRE         General Routing Encapsulation
 50     ESP         Encap Security Payload for IPv6
 51     AH          Authentication Header for Ipv6
 58     IPv6-ICMP   ICMP for Ipv6
```

  - Id: es un identificador que distingue cada paquete enviado por un host incrementándose de forma lineal. Además, se utiliza para identificar a los fragmentos de un mismo paquete ya que tendrán el mismo id. Cabe aclarar que no se encarga del orden de los fragmentos y para esto se usa un offset. Su formato es: `id:<number>;`
  - Geoip: brinda la posibilidad de poder emparejar la localización geográfica de una IP con un país determinado. Se puede detallar sí se quiere machear el origen y/o el destino. Ejemplo: `geoip: both, US, CA, UK;`
- Fragmentos:
  - Fragbits: es utilizado para modificar el mecanismo de fragmentación y para chequear si los bits reservados y de fragmentación fueron establecidos en el *header* de la IP. Cuando el paquete se desplaza a través de diferentes redes hasta llegar a destino puede suceder que el tamaño del paquete sea mayor al que la red puede soportar. En este caso, el paquete puede ser enviado en fragmentos.
  - Fragoffset: realiza un mapeo entre fragmentos de paquetes y valores decimales. Esto es utilizado en la etapa de re ensamblado para determinar que fragmentos pertenecen a que paquete y en qué orden.
- Protocolo de control de transmisión (*TCP*):
  - Seq: es usado en una firma para comprobar un número específico de secuencia TCP. Un número de secuencia es generado de manera aleatoria por cada uno de los dos extremos de una conexión TCP y es incrementado por cada byte que se envía. Cada número debe ser reconocido (*ack*) por el extremo opuesto de la conexión. A través de esta secuencia TCP se encarga del reconocimiento, orden y retransmisión. Formato: `seq:0;`
  - Ack: es el comprobante de recibo de todos los bytes previamente enviados por el otro extremo de la conexión TCP. Se utiliza en la firma para corroborar un número *ack* específico.
  - Window: es utilizado para comprobar un tamaño de ventana TCP específico. El tamaño de ventana TCP es un valor configurado por el receptor que indica el volumen de bytes que pueden ser recibidos sin hacer el *ack*. Es decir, que para que el emisor pueda enviar un nuevo volumen, primero se tiene que comprobar que se recibió el volumen anterior.
- Protocolo de mensajes de control de internet (*ICMP*): es un protocolo que ayuda al protocolo IP a ser más confiable ya que si bien no evita los problemas, puede ayudar a entender cuál fue el inconveniente y dónde. Posee cuatro partes que son las más importantes:
  - Itype: es usado para emparejar un tipo específico de ICMP. ICMP utiliza varias clases de mensajes y utiliza códigos para aclarar dichos mensajes. Los mensajes son distinguibles a través de distintos nombres, pero también a través de diferentes valores numéricos.

- Icode: es usado para emparejar un código específico de ICMP. El código es usado para aclarar el mensaje.
- Icmp_id: con esta palabra clave se puede machear valores específicos de identificación ICMP. Todo paquete ICMP recibe un id antes de ser enviado, al recibirlo el receptor envía un mensaje de respuesta con el mismo id para que sea identificado por el emisor.
- Icmp_seq: se utiliza para comprobar un numero de secuencia ICMP. Todos los mensajes ICMP poseen un numero de secuencia y su función es ayudar junto con el id a comprobar que respuesta corresponde a que solicitud.

Palabras clave de la carga útil (*Payload Keyword*)

- PCRE (*Perl Compatible Regular Expressions*): la palabra clave *pcre* es utilizada para crear reglas que contengan expresiones regulares y así poder realizar reglas menos específicas. Esto ayuda a que una misma regla sea utilizada para la detección de una mayor cantidad de malware y se comporte mejor ante sus posibles variantes.
  La complejidad que brinda *pcre* tiene como aspecto negativo un gran impacto en la performance. Por ello, para evitar que Suricata realice chequeos *pcre* en forma seguida se lo combina con *content*. En dicho caso, se realiza la comparativa con el contenido, y luego si el anterior tuvo un resultado negativo se realiza el chequeo a través de *pcre*.
  Formato de *pcre*: "`/<regex>/opts`";
  Ejemplo de *pcre*: `pcre:"/[0-9] {6}/";` en este caso ocurrirá un match si la carga útil contiene 6 números seguidos.
  Pcre posee varios aspectos que vienen con un valor predeterminado pero que pueden ser modificados ubicándolos detrás de *regex*, estos son:
  - i: para que no sea sensible a mayúsculas (lo será en caso de no especificarlo).
  - s: para que realice el chequeo de caracteres de nueva línea (sino los exceptuará).
  - m: puede hacer que una línea de la carga útil cuente como dos líneas.
- Patrón rápido (*Fast Pattern*): Si la palabra clave *fast pattern* es establecida en una regla, Suricata tomará el contenido previo a dicha palabra clave y lo usará como predeterminado en vez de seguir el orden explicitado. En el caso de la siguiente imagen se utilizará `content: "Badness"; distance:0;` en vez del ubicado en la parte superior.

  Esta palabra clave solo puede ser establecida una sola vez por cada regla y, en caso de no ser así es Suricata el que determina que contenido usará para realizar un match de patrón rápido en base a la siguiente lógica:
  1. Suricata primero identifica todo el contenido que hace match que contiene la prioridad más alta en la firma.
  2. Dentro de lo seleccionado en el punto anterior prioriza el contenido de mayor longitud de caracteres/bytes.
  3. Si hay varias opciones que matchean las condiciones anteriores se seleccionara el que posea la mayor diversidad de caracteres/bytes.
  4. En este caso se seleccionará el que posea un búfer (*list_id*) que fue registrado último.

5. En este caso se usará el ubicado primero (de izquierda a derecha) en la regla.

Es importante notar que si ocurre un match de un contenido positivo entonces el contenido negado será ignorado en la selección.

- Contenido: es una palabra clave muy importante en las firmas. Se coloca entre comillas lo que queremos que haga match y puede haber varios en una misma regla. El formato más simple es: `content: "……";`

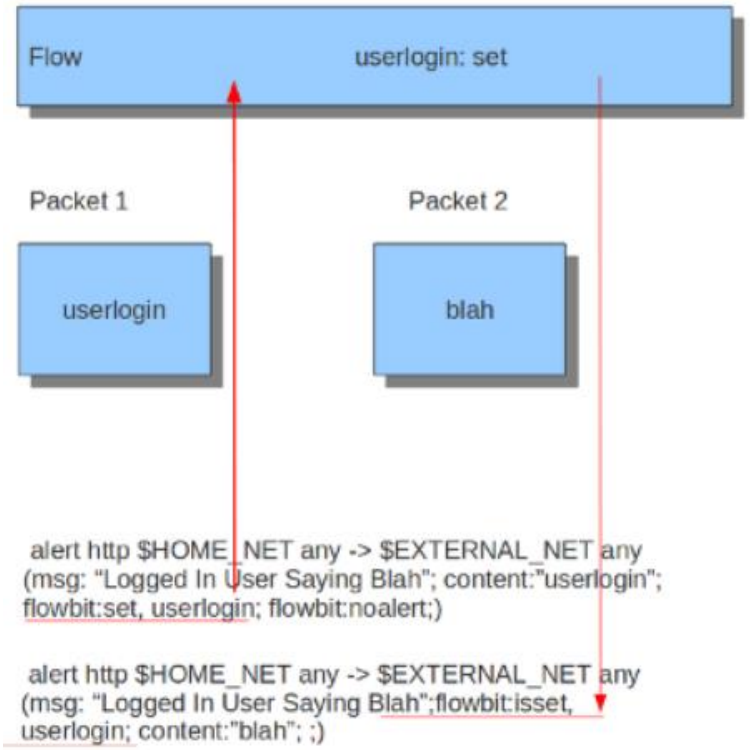
  Es importante destacar que *content* realiza match en bytes. Pueden ser usados cualquiera de los 256 valores disponibles, pero no todos los caracteres son imprimibles. Para estos, se utiliza notación hexadecimal escrita en mayúsculas. Por ejemplo:

```
|61| is a
|61 61| is aa
|41| is A
|21| is !
|0D| is carriage return
|0A| is line feed
```

  Es posible dejar que la firma realice un chequeo de toda la carga útil para realizar un match del contenido o permitir el chequeo en partes específicas de la carga útil.
- Nocase: esta palabra clave es un modificador de contenido que se utiliza cuando no queremos diferenciar entre caracteres en mayúscula y en minúscula. Se ubica detrás del contenido que se quiere modificar. Su formato es: `nocase;`
- Depth: modificador de contenido absoluto ubicado luego del contenido. Se debe especificar un valor que determina la cantidad de bytes que serán chequeados empezando desde el principio de la carga útil. Ejemplo: `depth: 12;`
- Offset: especifica que byte de la carga útil será chequeado para realizar un match. Por ejemplo, `offset: 3;` chequea a partir del cuarto byte.
- Distance: es un modificador relativo de contenido, esto quiere decir que existirá una relación entre lo especificado en el contenido y en el contenido siguiente. Se debe especificar un valor numérico (positivo o negativo) que determina el byte a partir del cual se realizara un chequeo para un match en relación con el match previo. *Distance* solo determina donde Suricata comienza a buscar un patrón. Por lo tanto, `distance: 5;` determina que el patrón puede estar en cualquier lugar luego del primer match + 5 bytes posteriores a este último. Se usa la palabra clave *within* para limitar el alcance del segundo match.
- Within: es un modificador relativo de contenido. Se debe especificar un valor numérico que no puede ser cero y dicho valor especifica la cantidad de bytes siguientes al primer match que se necesitan para realizar un match con el segundo contenido. Su uso se combina con *distance*.
- Isdataat: su propósito es el de comprobar si existen datos en una parte especifica de la carga útil. Se puede utilizar la opción *relative* para que el valor especificado se empiece a contar a partir del último match.
- Dsize: con esta palabra clave se puede realizar un match en el tamaño de la carga útil del paquete. Es conveniente para la detección de buffer overflows. Se puede utilizar para la búsqueda de tamaños de carga útil anormales. También para eliminar tráfico que no es de nuestro interés y así crear reglas más eficientes y exactas.

Palabras clave del flujo (*Flow Keywords*)

- Flowbits: consiste de una parte en la que se describe la acción a realizar y una segunda parte que contiene el nombre del flowbit. Todos los paquetes que pertenecen al mismo flujo son mantenidos en memoria por Suricata. Ver Ajustes de Flujo para más información. Flowbits puede hacer que se genere una alerta cuando dos o más paquetes diferentes hacen match. Para esto, Suricata debe conocer si el primer paquete hizo match luego de que el segundo lo haya hecho. Se utilizan marcas para identificar que ocurrió con el primer paquete. Por ejemplo:



  Al ver la primera regla se interpreta que al ocurrir un match se generaría una alerta, pero este no es el caso debido a la presencia de *flowbits: noalert* al final de la misma. Por lo tanto, el propósito de esta regla es el de chequear un match en *userlogin* y realizar una marca en el flujo. La segunda regla no tiene ningún sentido sin la primera. Con la segunda regla se puede chequear si se cumplió o no la primera condición. Si ambas reglas hicieron match, se generará una alerta.

Palabras clave de archivo (*File Keywords*)

- Nombre del archivo (*filename*): realiza un match en el nombre del archivo. Ejemplo: `filename:"secret";`
- Extensión del archivo (*fileext*): realiza un match en la extensión del archivo. Ejemplo: `fileext:"jpg";`
- Almacenamiento del archivo (*filestore*): almacena el archivo si ocurre un match. Sintaxis: `filestore: <direction>, <scope>;`
- Suma de comprobación del archivo (filemd5): realiza un match entre la suma de comprobación del archivo y una lista de sumas de comprobación (checksums). Ejemplo: `filemd5: md5-blacklist;` ó `filemd5: !md5-whitelist;`

El nombre del archivo deberá incluir la ruta de su ubicación. De lo contrario se lo buscará en */etc/suricata/rules/filename*. Se utiliza el signo de exclamación para negar, esto permite listas blancas.

- Tamaño del archivo (*filesize*): realiza un match en el tamaño del archivo (en bytes) mientras está siendo transferido. Sintaxis: `filesize:<value>;`

3.4: Gestión de reglas

- Gestión de reglas con Oinkmaster

  Oinkmaster es un programa que se utiliza para descargar e instalar reglas. Fue publicado bajo licencia BSD.

  Realizamos la instalación de Oinkmaster.

```
ale@ubuntu:~$ sudo apt-get install oinkmaster
```

  Se puede utilizar varios conjuntos de reglas como *Emerging Threats* (ET), Emerging Threats Pro y VRT. En este caso se usará Emerging Threats. Este es un centro de investigación abierto de seguridad que produce fuentes de datos a partir de nuevas amenazas. Lo que hace que este programa sea tan eficaz es su espíritu colaborativo ya que cualquier persona se puede encargar de aportar nuevas ideas y de realizar revisiones del contenido. El principal objetivo es hacer que este proceso ocurra de forma rápida y abierta para ayudar a todos los profesionales de la seguridad a responder en el menor tiempo posible tanto ante amenazas conocidas como desconocidas.

  Para utilizar el mencionado conjunto de reglas, Oinkmaster debe saber dónde ubicarlas. Entonces accedemos al archivo de configuración de Oinkmaster.

```
ale@ubuntu:~$ sudo nano /etc/oinkmaster.conf
```

  Luego se añade la dirección correspondiente al conjunto de reglas y se guarda el archivo.

```
  GNU nano 2.5.3                                File: /etc/oinkmaster.conf

# URL examples follows. Replace <oinkcode> with the code you get on the
# Snort site in your registered user profile.

# VRT certified rules for registered users, Snort 2.9.
# url = http://www.snort.org/pub-bin/oinkmaster.cgi/<oinkcode>/snortrules-snapshot-2.9.tar.gz

# VRT certified rules for registered users, Snort 2.7.
# url = http://www.snort.org/pub-bin/oinkmaster.cgi/<oinkcode>/snortrules-snapshot-2.7.tar.gz

# VRT certified rules for registered users, Snort 2.8.
# url = http://www.snort.org/pub-bin/oinkmaster.cgi/<oinkcode>/snortrules-snapshot-2.8.tar.gz

# VRT certified rules for registered users, Snort-CURRENT
# ("CURRENT" here means experimental snapshots!).
# url = http://www.snort.org/pub-bin/oinkmaster.cgi/<oinkcode>/snortrules-snapshot-CURRENT.tar.gz


# Community rules and Snort 2.4.
# url = http://www.snort.org/pub-bin/downloads.cgi/Download/comm_rules/Community-Rules-2.4.tar.gz

# Community rules for snort-CURRENT
# url = http://www.snort.org/pub-bin/downloads.cgi/Download/comm_rules/Community-Rules-CURRENT.tar.gz


# Example for rules from the Emerging Threats site (previously known as Bleeding Snort).
url= https://rules.emergingthreats.net/open/suricata-3.2/emerging.rules.tar.gz
# url = http://www.emergingthreats.net/rules/emerging.rules.tar.gz
# Old url:
# url = http://www.bleedingsnort.com/downloads/bleeding.rules.tar.gz
```

  Creamos una carpeta que contendrá las reglas nuevas.

```
ale@ubuntu:~$ sudo mkdir /etc/suricata/rules/
```

Luego descargamos nuestras reglas y configuramos Oinkmaster para que las use en lugar de las predeterminadas.

```
ale@ubuntu:/etc$ sudo oinkmaster -C /etc/oinkmaster.conf -o /etc/suricata/rules
```

Dentro de la carpeta rules habrá tres archivos de configuración:

- *classification.config*: contiene información para priorizar reglas. Cada clasificación agrupa un conjunto de reglas para las cuales posee un nombre breve, una descripción y una prioridad predeterminada. Cada regla hereda su prioridad de la clase y cada regla puede sobrescribir su prioridad con el campo *priority*, ubicado en su cuerpo.

```
#
# config classification:shortname,short description,priority
#

#Traditional classifications. These will be replaced soon

config classification: not-suspicious,Not Suspicious Traffic,3
config classification: unknown,Unknown Traffic,3
config classification: bad-unknown,Potentially Bad Traffic, 2
config classification: attempted-recon,Attempted Information Leak,2
config classification: successful-recon-limited,Information Leak,2
config classification: successful-recon-largescale,Large Scale Information Leak,2
config classification: attempted-dos,Attempted Denial of Service,2
config classification: successful-dos,Denial of Service,2
config classification: attempted-user,Attempted User Privilege Gain,1
config classification: unsuccessful-user,Unsuccessful User Privilege Gain,1
config classification: successful-user,Successful User Privilege Gain,1
config classification: attempted-admin,Attempted Administrator Privilege Gain,1
config classification: successful-admin,Successful Administrator Privilege Gain,1
config classification: rpc-portmap-decode,Decode of an RPC Query,2
config classification: shellcode-detect,Executable Code was Detected,1
config classification: string-detect,A Suspicious String was Detected,3
config classification: suspicious-filename-detect,A Suspicious Filename was Detected,2
```

- *reference.config*: contiene direcciones de páginas web que pertenecen a los sitios de donde se obtienen las firmas de malware y demás.

```
# config reference: system URL

config reference: bugtraq   http://www.securityfocus.com/bid/
config reference: bid       http://www.securityfocus.com/bid/
config reference: cve       http://cve.mitre.org/cgi-bin/cvename.cgi?name=
#config reference: cve       http://cvedetails.com/cve/
config reference: secunia   http://www.secunia.com/advisories/

#whitehats is unfortunately gone
config reference: arachNIDS http://www.whitehats.com/info/IDS

config reference: McAfee    http://vil.nai.com/vil/content/v_
config reference: nessus    http://cgi.nessus.org/plugins/dump.php3?id=
config reference: url       http://
config reference: et        http://doc.emergingthreats.net/
config reference: etpro     http://doc.emergingthreatspro.com/
config reference: telus     http://
config reference: osvdb     http://osvdb.org/show/osvdb/
config reference: threatexpert http://www.threatexpert.com/report.aspx?md5=
config reference: md5       http://www.threatexpert.com/report.aspx?md5=
config reference: exploitdb http://www.exploit-db.com/exploits/
config reference: openpacket https://www.openpacket.org/capture/grab/
config reference: securitytracker http://securitytracker.com/id?
config reference: secunia   http://secunia.com/advisories/
config reference: xforce    http://xforce.iss.net/xforce/xfdb/
config reference: msft      http://technet.microsoft.com/security/bulletin/
```

- *threshold.config*: archivo que se utiliza para configurar el umbral y así reducir la cantidad de falsas alarmas. También se puede utilizar para escribir una nueva

generación de reglas. Los comandos de umbral limitan el número de veces que un evento determinado se registra durante un intervalo de tiempo especificado.

```
# The syntax is the following:
#
# threshold gen_id <gen_id>, sig_id <sig_id>, type <limit|threshold|both>, track <by_src|by_dst>, count <n>, seconds <t>
#
# event_filter gen_id <gen_id>, sig_id <sig_id>, type <limit|threshold|both>, track <by_src|by_dst>, count <n>, seconds <t>
#
# suppress gen_id <gid>, sig_id <sid>
# suppress gen_id <gid>, sig_id <sid>, track <by_src|by_dst>, ip <ip|subnet>
#
# The options are documented at https://redmine.openinfosecfoundation.org/projects/suricata/wiki/Global-Thresholds
#
# Please note that thresholding can also be set inside a signature. The interaction between rule based thresholds
# and global thresholds is documented here:
# https://redmine.openinfosecfoundation.org/projects/suricata/wiki/Global-Thresholds#Global-thresholds-vs-rule-thresholds

# Limit to 10 alerts every 10 seconds for each source host
#threshold gen_id 0, sig_id 0, type threshold, track by_src, count 10, seconds 10

# Limit to 1 alert every 10 seconds for signature with sid 2404000
#threshold gen_id 1, sig_id 2404000, type threshold, track by_dst, count 1, seconds 10

# Avoid to alert on f-secure update
# Example taken from http://blog.inliniac.net/2012/03/07/f-secure-av-updates-and-suricata-ips/
#suppress gen_id 1, sig_id 2009557, track by_src, ip 217.110.97.128/25
#suppress gen_id 1, sig_id 2012086, track by_src, ip 217.110.97.128/25
#suppress gen_id 1, sig_id 2003614, track by_src, ip 217.110.97.128/25
```

Debemos modificar el archivo de configuración de Suricata para agregar la ruta a los mismos.

Accedemos al archivo de configuración de Suricata.

```
ale@ubuntu:~$ sudo nano /etc/suricata/suricata.yaml
```

Añadimos las nuevas rutas, comentamos las predeterminadas y guardamos los cambios.

```
  GNU nano 2.5.3                                File: /etc/suricata/suricata.yaml

 - emerging-trojan.rules
 - emerging-user_agents.rules
 - emerging-voip.rules
 - emerging-web_client.rules
 - emerging-web_server.rules
# - emerging-web_specific_apps.rules
 - emerging-worm.rules
 - tor.rules
# - decoder-events.rules # available in suricata sources under rules dir
# - stream-events.rules  # available in suricata sources under rules dir
 - http-events.rules    # available in suricata sources under rules dir
 - smtp-events.rules    # available in suricata sources under rules dir
 - dns-events.rules     # available in suricata sources under rules dir
 - tls-events.rules     # available in suricata sources under rules dir
# - modbus-events.rules  # available in suricata sources under rules dir
# - app-layer-events.rules  # available in suricata sources under rules dir
# - dnp3-events.rules       # available in suricata sources under rules dir

#classification-file: /etc/suricata/classification.config
#reference-config-file: /etc/suricata/reference.config
#threshold-file: /etc/suricata/threshold.config
classification-file: /etc/suricata/rules/classification.config
reference-config-file: /etc/suricata/rules/reference.config
threshold-file: /etc/suricata/rules/threshold.config

##
## Step 3: select outputs to enable
##

# The default logging directory.  Any log or output file will be
# placed here if its not specified with a full path name. This can be
# overridden with the -l command line parameter.
default-log-dir: /var/log/suricata/
```

Corroboramos que todo funciona al ejecutar Suricata, para esto especificamos la ruta del archivo de configuración con –c y la interfaz de la palca de red con –i.

```
ale@ubuntu:~$ sudo suricata -c /etc/suricata/suricata.yaml -i wlx002191960e38
6/5/2017 -- 14:18:49 - <Info> - Shortening device name to: wlx0..0e38
6/5/2017 -- 14:18:49 - <Notice> - This is Suricata version 3.2.1 RELEASE
6/5/2017 -- 14:18:56 - <Notice> - all 4 packet processing threads, 4 management threads initialized, engine started.
```

En base a lo anterior, se podría dar el caso en que Suricata intente cargar archivos de reglas que no se encuentren en la carpeta especificada. Este aviso se mostrará mediante un *warning*. Para que este último desaparezca se deben desactivar los archivos en cuestión colocando un # delante en el archivo de configuración *suricata.yaml*.
Para frenar la ejecución de Suricata presionamos *ctrl + c*.
El conjunto de reglas Emerging Threats posee una mayor cantidad de reglas que las que fueron cargadas en Suricata. Para ver cuáles son las reglas disponibles ingresamos:

```
ale@ubuntu:~$ ls /etc/suricata/rules/*.rules
```

Luego, si queremos añadir alguna regla solo debemos agregarla a un archivo de reglas y añadir la ruta a este último al archivo de configuración *suricata.yaml*. Contrariamente, si queremos quitar una regla debemos ir al archivo de reglas de la contiene y eliminar o comentar con # la línea que lo invoca. Para quitar una regla de forma permanente, debemos buscar la identificación de la firma (*sid)* en la carpeta rules para luego añadir el comando *disablesid <sid1>,<sid2>, …,<sidn>* en el archivo de configuración *oinkmaster.conf*.
Además, Oinkmaster posee reglas que están desactivadas de forma predeterminada. Para añadirlas deberíamos no solo añadir la ruta del archivo que las contiene al archivo de configuración de Suricata, sino también habrá que agregar su *sid* al archivo de configuración de Oinkmaster añadiendo el comando *enablesid: <sid1>,<sid2>, …,<sidn>.*
Oinkmaster también nos brinda la posibilidad de modificar reglas. Por ejemplo, si se utiliza Suricata como un IPS/en línea y queremos modificar una regla que emite una alerta cuando ocurre una coincidencia para que además ahora se deshaga del paquete debemos modificar el archivo de configuración de Oinkmaster y agregar *drop* a la regla en la sección *modifysid*. Luego se debe reiniciar Oinkmaster para aplicar los cambios.
Ver http://doc.emergingthreats.net/bin/view/Main/

- Agregando nuestras propias reglas

  Empezamos creando un archivo para nuestra nueva regla.

  ```
  ale@ubuntu:/etc/suricata/rules$ sudo nano local.rules
  ```

  Escribimos la regla, para esto debemos referirnos a la sección 3.3.
  Agregamos la regla al archivo de configuración *suricata.yaml* y ejecutamos Suricata para corroborar que la regla fue cargada correctamente. Por último, podemos chequear que la regla funciona realizando acciones que la disparen. Es recomendable enviar la salida a un archivo de registro para ver la respuesta obtenida.

  ```
  ale@ubuntu:~$ sudo tail -f /var/log/suricata/fast.log
  ```

- Actualización de reglas

  Es importante tener en cuenta que las reglas tienen que ser actualizadas en forma periódica. Esto se realiza con el siguiente comando:

  ```
  ale@ubuntu:/etc$ sudo oinkmaster -C /etc/oinkmaster.conf -o /etc/suricata/rules
  ```

- Recarga de reglas

  Suricata puede recargar sus reglas sin tener que reiniciar el IDS. Esto puede realizarse mediante una señal que se envía a Suricata: `kill –USR2 $(pidof suricata)`, o a través de un socket unix: `suricatasc –c reload-rules.`

  Cuando se le envía la señal a Suricata para que recargue las reglas ocurren los siguientes pasos:

  1. Carga la nueva configuración.
  2. Carga las nuevas reglas.
  3. Construye el nuevo motor de detección.
  4. Intercambia el viejo motor de detección por el nuevo.
  5. Asegura que todos los hilos de ejecución están actualizados.
  6. Libera el viejo motor de detección.

  Suricata continuará el procesamiento de los paquetes con normalidad durante el mencionado proceso. De todos modos, hay que tener en cuenta que el sistema debe poseer suficiente memoria para ambos motores de detección.

3.5: Rendimiento

Suricata vs Snort

Análisis han mostrado que Suricata posee una mayor tasa de precisión que Snort, aunque esto es una consecuencia de requerir una mayor demanda relativa en el CPU. Los resultados revelaron que, debido a la utilización de varios núcleos de manera más uniforme, Suricata tiene el potencial de ser más escalable y eficiente donde hay más núcleos disponibles. Sin embargo, el requerir una mayor cantidad de recursos hace que la precisión de Suricara disminuya en ambientes de desarrollo donde los mismos están limitados.

Modos de ejecución

Suricata está constituido por distintos bloques llamados hilos de ejecución, módulos de hilos y colas. Un hilo es un tipo de proceso que se ejecuta en una computadora. Como mencionamos con anterioridad, Suricata tiene la propiedad de ser multi-hilado, es decir que múltiples hilos de ejecución están activos al mismo tiempo. Un módulo de hilo puede componer distintas funcionalidades. Por ejemplo, un módulo realiza la decodificación de paquetes, otro es el encargado de realizar la detección y otro puede ser el encargado de generar las salidas. Un paquete puede ser procesado por más de un hilo, en dicho caso, será enviado al próximo hilo a través de una cola. Los paquetes son procesados por un solo hilo al mismo tiempo, pero muchos paquetes son procesados al mismo tiempo por el motor. Un hilo de ejecución puede poseer varios módulos de hilos, pero solo puede estar activo uno al mismo tiempo.

Se puede optar por diferentes modos de funcionamiento predefinidos. Para mostrar todos los modos disponibles:

```
ale@ubuntu:~$ suricata --list-runmodes
```

Los modos de ejecución personalizados son: *auto*, *single*, *autofp* y *workers*. El modo *autofp* es más apropiado si el método de captura no soporta múltiples lectores, como ocurre con el método *pcap*. Por lo tanto, en este caso, *autofp* realizara un balance de carga de manera automática teniendo en cuenta los *CPU's* disponibles. Los schedulers soportados son: *round-robin*, *active-packets* (los flujos se asignan a hilos que poseen la menor cantidad de paquetes no procesados) y *hash* (el más similar a un método aleatorio).

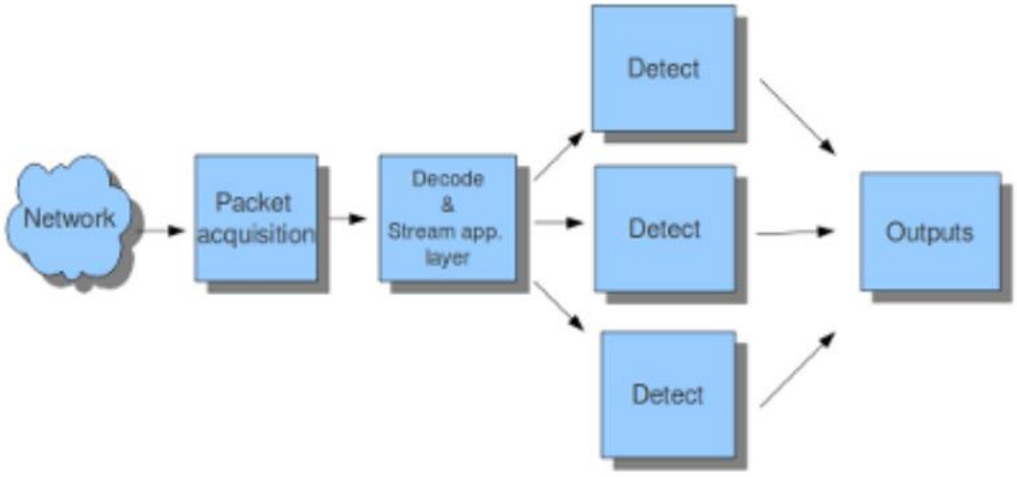
El modo *workers* tiene la ventaja de mantener todo el procesamiento del paquete en un hilo de ejecución, además si se establece afinidad al *CPU* todo el procesamiento también se mantendrá en un único núcleo de *CPU*. En cambio, en *autofp* se realiza una contención en las colas que transfieren paquetes desde el hilo que se encarga de la lectura hacia los hilos que se encargan de la detección.

En ambos casos, los hilos acceden a estructuras globales de datos como la tabla de flujo, la tabla de host, etc.

Con el modo *workers* junto con la afinidad activada nos aseguramos que un paquete sea procesado por un único núcleo de *CPU*, pero esto no resuelve el problema de enlace entre el *CPU* que recibe el paquete y el usado en Suricata. Para resolver dicho problema, debemos asegurar que cuando un paquete es recibido en una cola, el *CPU* que realiza el manejo del paquete sea el mismo que procese el paquete en Suricata. Esto se realiza estableciendo el método de captura en *AF_PACKET*.

La tarea que más recurso insume es la detección ya que cada paquete será chequeado contra miles de firmas.

Ejemplo del modo de funcionamiento predeterminado *autofp*:



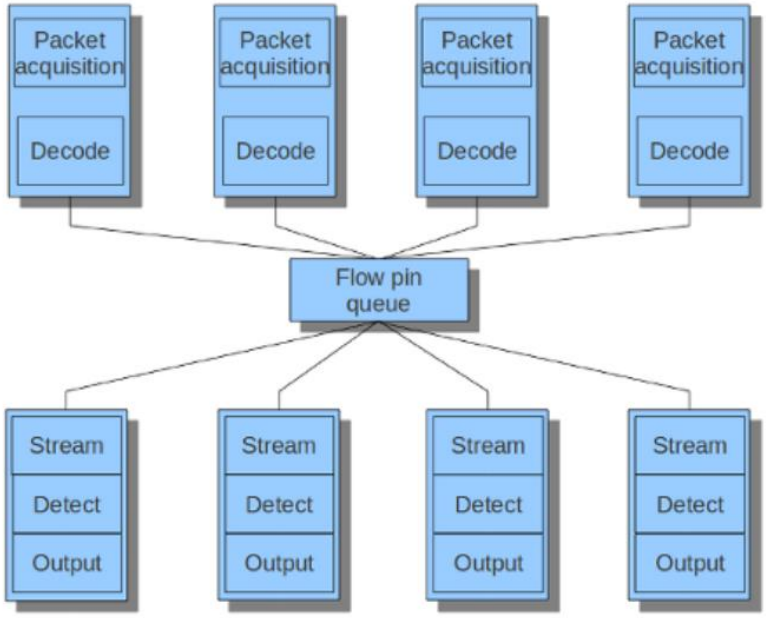
En nuestro caso, se ejecutará Suricata en el modo mostrado con anterioridad. Por lo tanto, se utilizarán los dos núcleos disponibles para ejecutar un hilo en cada uno sumado al uso de *hyperthreading* por lo que se deben añadir dos hilos. Es por esto que al ejecutar Suricata en modo IDS podemos visualizar que los cuatro hilos son utilizados para realizar tareas de administración la cual incluye las tareas de recepción, decodificación, stream, detección, verdict, rechazo y generación de la salida.

```
1/11/2017 -- 13:18:31 - <Notice> - This is Suricata version 3.2.1 RELEASE
1/11/2017 -- 13:19:12 - <Notice> - all 4 packet processing threads, 4 management threads initialized, engine started.
```

En cambio, en el modo IPS se tiene un hilo extra que con el *hyperthreading* hace que sean dos dedicados a la tarea de verdict que comunica al IPS con el kernel.

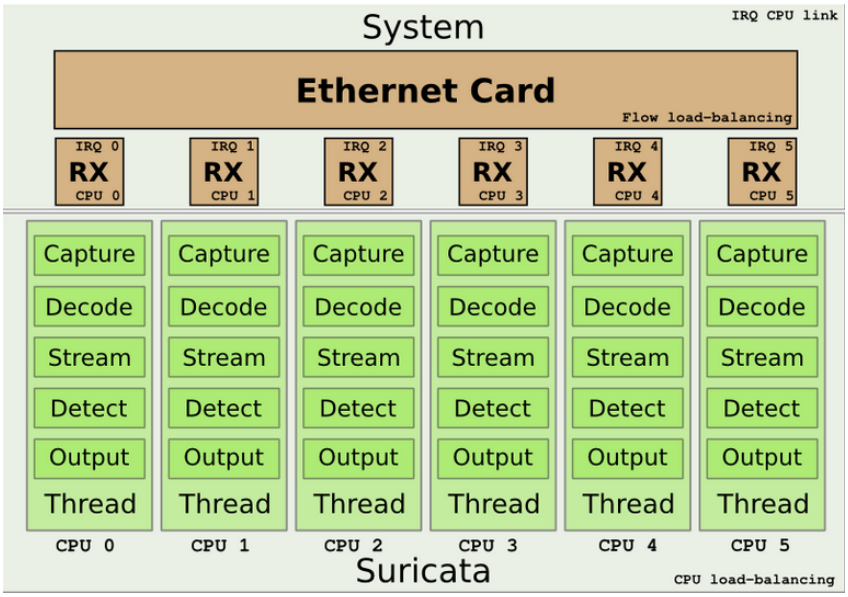
```
1/11/2017 -- 13:35:31 - <Notice> - This is Suricata version 3.2.1 RELEASE
1/11/2017 -- 13:36:12 - <Notice> - all 6 packet processing threads, 4 management threads initialized, engine started.
```

Si nos centramos en la adquisición de paquetes se puede utilizar una librería llamada *pfring* que mejora el rendimiento de captura al compararlo con *libcap.* Se realiza el balanceo de la carga basado en el flujo y cada uno de estos sigue su propia ruta.



Captura de paquetes

- Balance de carga: para obtener la mejor performance Suricata debe ejecutarse en el modo '*workers*'. Esto significa que habrá múltiples hilos, en los que cada uno estará corriendo con el pipeline lleno de paquetes y recibiendo los mismos desde el método de captura (*AF_PACKET* o *PF_RING*). Esto significa que este método es el encargado de distribuir los paquetes entre los diferentes hilos de ejecución. Un aspecto importante a tener en cuenta es que Suricata necesita obtener ambos lados de un flujo en el mismo hilo y en el orden correcto.
  En la siguiente imagen podemos ver que cada *CPU* tiene asociada una interrupción (*IRQ)*, la cual es lanzada para realizar la recepción (*RX*) de un paquete. Luego, el paquete es procesado por el hilo de ejecución que está asociado al *CPU*.



- Escalado lateral de recepción: el escalado lateral de recepción (o **RSS** de sus siglas en inglés *Receive Side Scaling*) es una técnica utilizada por las placas de red para distribuir el tráfico entrante a la tarjeta de interfaz de red entre varias colas. Está destinado a mejorar la performance, aunque es importante destacar que fue diseñado para tráfico normal y no para un escenario de captura de paquetes por parte de un IDS. RSS utiliza un algoritmo de hash para distribuir el trafico entrante entre varias colas. Dicho hash normalmente no es simétrico. Esto quiere decir que cuando se reciben ambos lados de un flujo, cada uno podría terminar en una cola diferente. Lamentablemente, al implementar Suricata este es un escenario común cuando se utilizan puertos *span* o *taps*.

<u>Consideraciones para la puesta a punto</u>

Configuraciones a verificar en el archivo *suricata.yaml* para lograr un rendimiento óptimo:

- ***max-pending-packets: <number>***; controla el número de paquetes que el motor puede manejar de manera simultánea, manteniéndolos en memoria. Estableciendo valores más altos generalmente mantiene los hilos más ocupados, pero establecerlo demasiado alto producirá degradación.
  Valor sugerido: 1000 o superior. Valor configurado: 1024. Máximo: 65000.
  Se encuentra en la línea 934 del archivo de configuración de Suricata.
- ***mpm-algo: <ac|hs|ac-bs|ac-ks>***; controla el algoritmo de coincidencia de patrones (*pattern matching*). El valor predeterminado es *auto* en el que utilizara *hs* si *Hyperscan* está disponible o *ac* en caso contrario. Valor configurado: auto.
  Se encuentra en la línea 1340 del archivo de configuración de Suricata.
- ***detect.profile: <low|medium|high|custom>***; el motor de detección agrupa las firmas para que el paquete sea revisado solo por las firmas con las que en realidad podría machear. La configuración del perfil establece cuan agresivo es el agrupado. Más alto es mejor, aunque resulta en una utilización de memoria mucho mayor. La opción *custom* permite modificar el tamaño de los grupos:

```
custom-values:
  toclient-groups: 50
  toserver-groups: 50
```

Se encuentra en la línea 1280 del archivo de configuración de Suricata.

- ***detect.sgh-mpm-context: <auto|single|full>***; el emparejador de múltiples patrones puede poseer un contexto por grupo de firmas (*full*) o uno global (*single*). La opción *auto* selecciona entre *single* y *full* en base al *mpm-algo* seleccionado; *ac* y *ac-bs* usa *single*. Los demás usan *full*. Si lo configuramos en *full* con *ac* se requerirá más de 32GB de memoria para un conjunto de reglas razonable.
  El valor configurado es *auto*.
  Se encuentra en la línea 1285 del archivo de configuración de Suricata.

Hyperscan

Es una biblioteca de concordancia de múltiples expresiones regulares de alto rendimiento desarrollada por Intel. Utiliza técnicas de autómatas híbridos[7] para poder realizar match de grandes números de expresiones regulares en forma simultánea, así como la coincidencia de expresiones regulares a través de flujos de datos.

En Suricata es utilizado para llevar a cabo *Pattern Matching Multiple* (*MPM*).

- Instalación del soporte *Hyperscan* versión 4.4.1 en Ubuntu:
  Primero debemos instalar las dependencias:
  Añadimos *Ragel*, el cual es un compilador de estados finitos y un generador parser.

```
ale@ubuntu:~$ sudo apt-get install cmake ragel
```

  Otra dependencia es el paquete que contiene los archivos de desarrollo de librerías C++.

```
ale@ubuntu:~$ sudo apt-get install libboost-dev
```

  Agregamos los *headers* y las librerías estáticas de Python junto con una librería compresora de archivos de desarrollo.

```
ale@ubuntu:~$ sudo apt-get install python-dev libbz2-dev
```

  Instalamos el conjunto de bibliotecas *Boost*, las cuales nos permiten extender las capacidades del lenguaje de programación C++.
  (http://downloads.sourceforge.net/project/boost/boost/1.60.0/boost_1_60_0.tar.gz)

```
ale@ubuntu:~$ wget http://downloads.sourceforge.net/project/boost/boost/1.60.0/boost_1_60_0.tar.gz
```

```
ale@ubuntu:~$ tar xvzf boost_1_60_0.tar.gz
```

```
ale@ubuntu:~$ cd boost_1_60_0
```

```
ale@ubuntu:~/boost_1_60_0$ ./bootstrap.sh --prefix=~/tmp/boost-1.60
```

[7] Modelo matemático para describir sistemas de manera precisa en los que procesos computacionales digitales interactúan con procesos físicos analógicos. Es una máquina de estados finitos que contiene un conjunto finito de variables cuyos valores son descriptos a través de un conjunto de ecuaciones diferenciales ordinarias.

```
ale@ubuntu:~/boost_1_60_0$ ./b2 install
```

Instalamos el generador de documentación *Doxygen*.

```
ale@ubuntu:~$ sudo apt-get install doxygen
```

SQLite es un sistema de gestión de bases de datos relacional.

```
ale@ubuntu:~$ sudo apt-get install sqlite3 libsqlite3-dev
```

Generador de documentación Python de código abierto *Sphinx*.

```
ale@ubuntu:~$ sudo apt-get install python-sphinx
```

Una vez que las dependencias están instaladas descargamos *Hyperscan*.

```
ale@ubuntu:~$ git clone https://github.com/01org/hyperscan
```

```
ale@ubuntu:~$ cd hyperscan/
```

```
ale@ubuntu:~/hyperscan$ mkdir build
```

```
ale@ubuntu:~/hyperscan$ cd build/
```

```
ale@ubuntu:~/hyperscan/build$ cmake -DBUILD_STATIC_AND_SHARED=1 ../
```

Creamos el fichero ejecutable para su instalación con el comando *make*.

Realizamos la instalación del fichero ejecutable con *sudo make install*. Al finalizar deberíamos ver algo similar a lo siguiente.

```
Install the project...
-- Install configuration: "RELWITHDEBINFO"
-- Installing: /usr/local/lib/pkgconfig/libhs.pc
-- Installing: /usr/local/include/hs/hs.h
-- Installing: /usr/local/include/hs/hs_common.h
-- Installing: /usr/local/include/hs/hs_compile.h
-- Installing: /usr/local/include/hs/hs_runtime.h
-- Installing: /usr/local/lib/libhs_runtime.a
-- Installing: /usr/local/lib/libhs_runtime.so.4.4.1
-- Installing: /usr/local/lib/libhs_runtime.so.4
-- Installing: /usr/local/lib/libhs_runtime.so
-- Installing: /usr/local/lib/libhs.a
-- Installing: /usr/local/lib/libhs.so.4.4.1
-- Installing: /usr/local/lib/libhs.so.4
-- Installing: /usr/local/lib/libhs.so
```

Ahora solo nos resta agregar la ruta */usr/local/lib* para que *ld* la incluya en la búsqueda.

```
ale@ubuntu:~/hyperscan/build$ echo "/usr/local/lib" | sudo tee --append /etc/ld.so.conf.d/usrlocal.conf
```

```
ale@ubuntu:~/hyperscan/build$ sudo ldconfig
```

Luego de instalar Suricata podremos corroborar si *Hyperscan* está correctamente integrado ejecutando el siguiente comando:

```
ale@ubuntu:~$ suricata --build-info|grep Hyperscan
  Hyperscan support:                       yes
ale@ubuntu:~$
```

Configuración de alto rendimiento

Si se quiere sacar el mayor provecho al hardware disponible, es decir, utilizar una mayor cantidad de RAM y exprimir al máximo al procesador se deben tener en cuenta diversas opciones en el archivo de configuración *suricata.yaml*.

La primera de ellas tiene que ver con el perfil utilizado. Este fue explicado aquí.

Procedemos a configurar el perfil como ilustra la siguiente imagen.

```
detect:
  profile: custom
  custom-values:
      toclient-groups: 200
      toserver-groups: 200
  sgh-mpm-context: auto
  inspection-recursion-limit: 3000
```

Cabe aclarar que a la hora de mejorar el rendimiento es más importante poseer un procesador más potente en lugar de una mayor cantidad de memoria RAM.

Ignorando tráfico

En muchos casos existen razones para ignorar un determinado tipo de tráfico de datos, por ejemplo, cuando se trata de una red, un host o un sitio confiable. Para llevarlo a cabo existen determinadas estrategias:

- Filtros de captura (BPF): a través de este filtro, los métodos de captura *pcap*, *af-packet* y *pf_ring* pueden ser configurados para indicar qué enviar a Suricata y qué no. Por ejemplo, un filtro *tcp* simple solo enviará paquetes *tcp*.
- Reglas de paso por alto: se trata de reglas de Suricata que al coincidir produce que el paquete sea pasado por alto, y en el caso de TCP el resto del flujo. Son similares a una regla normal excepto que en vez de empezar con *alert* o *drop* lo hacen con *pass*. A diferencia de los filtros de captura, logueos como *http.log* se generan todavía por este tráfico.
- Supresión: reglas de supresión pueden ser utilizadas para asegurar que no se generan alertas con respecto a un host. Sin embargo, esto no es muy eficiente ya que la supresión solo se considera luego de ocurrida la coincidencia. Es decir que Suricata primero inspecciona una regla, y luego considerará las supresiones del host.

Tcmalloc

Es una librería de Google que forma parte de la suite google-perftools para mejorar el manejo de memoria en los hilos de ejecución de un programa. Es simple de usar y trabaja bien con Suricata. Conduce a menores aceleraciones de procesamiento y reduce el uso de memoria.

- Instalación.
  ```
  ale@ubuntu:~$ sudo apt-get install libtcmalloc-minimal4
  ```
- Modo de uso: para utilizar *tcmalloc* primero debemos precargarlo a la ejecución de Suricata de la siguiente manera:

```
ale@ubuntu:~$ sudo LD_PRELOAD="/usr/lib/libtcmalloc_minimal.so.4" suricata -c /etc/suricata/suricata.yaml -i ens33
```

3.6: Configuración: Suricata.yaml

Max-pending-packets

Modos de ejecución

Default-packet-size

Con esta opción se puede establecer el tamaño de los paquetes en la red. Es posible que a veces se tengan que procesar paquetes de mayor tamaño, en dicho caso el motor podrá hacerlo, aunque su procesamiento tendrá como consecuencia una disminución de la performance. Su configuración se realiza en base a la unidad máxima de transferencia (MTU) y al tipo de hardware. Ejemplo, `default-packet-size: 1514`

Orden de acción

Resultados a través de eventos como salida

- Registro por defecto: Todas las salidas (alertas y eventos) que genera Suricata se almacenan de forma predeterminada en la carpeta */var/log/suricata*. Esto puede encontrarse en la siguiente línea: `default-log-dir: /var/log/suricata`

  Esta carpeta puede ser sobrescrita a través de la opción *–l* como parámetro al ejecutar Suricata. Por ejemplo: `suricata -c suricata.yaml -i eth0 -l /var/log/suricata-logs/`

  O a través del cambio de carpeta en el archivo de configuración *yaml*.

- Salidas: la estructura general de la salida es la siguiente.

  ```
  outputs:
   -fast:
      enabled: yes
      filename: fast.log
      append: yes/no
  ```

  Si se habilitan todos los registros resultará en un rendimiento mucho menor y en el uso de mayor espacio en disco. Por lo tanto, la recomendación es habilitar solo los log necesarios. Los registros disponibles son:
  - Línea basada en el registro de alertas (*fast.log*): contiene alertas de una línea individual que le facilita la identificación y lectura a las personas.
  - Eve (*Extensible Event Format*): se trata de un archivo de salida JSON que contiene eventos y alertas. Permite la integración con herramientas de terceros.
  - Alerta de salida para utilizar con Barnyard (*unified2.alert*): el formato de este registro es compatible con la salida *unified2* correspondiente a otro IDS popular.
  - Línea basada en el registro de solicitudes HTTP (*http.log*): este registro realiza un seguimiento de todos los eventos basados en tráfico HTTP.

- Línea basada en el registro de consultas y respuestas DNS (*dns.log*): realiza un seguimiento de todos los eventos DNS, es decir tanto consultas como respuestas.
- Registro de paquetes (*pcap-log*): esta opción permite guardar todos los paquetes registrados por Suricata.
- Registro de alertas detalladas (*alert-debug.log*): este tipo de registro brinda información suplementaria sobre una alerta. Es recomendada para personas que investigan falsos positivos o desarrollan firmas.
- Alert output to prelude (*alert-prelude*): las alertas prelude contienen una gran cantidad de campos y son utilizadas en conjunto con el administrador prelude IDS.
- Estadísticas (stats.log): se pueden configurar diversas opciones referidas al registro de estadísticas.
- Syslog: esta opción hace posible el envío de todos los eventos y alertas a un registro de sistema.
- Línea basada en el registro de paquetes soltados (*drop.log*): todos los paquetes soltados en base a determinadas reglas son guardados en este registro.

Motor de detección

- Configuración de inspección: el motor de detección es el encargado de construir grupos de firmas internos. Luego, Suricata carga las firmas que utilizará para comparar con el tráfico de red. Muchas veces sucede que ciertas reglas no son necesarias, por ejemplo, si se trata de un paquete que posee el protocolo UDP todas las firmas del protocolo TCP no serán útiles. Por esta razón las firmas se dividen en grupos. Para evitar la excesiva cantidad de grupos, que redundará en un gran uso de memoria, muchas firmas se agrupan en base a propiedades en común. Tampoco se obtiene un beneficio si la cantidad de grupos es pequeña, ya que el rendimiento será menor. Para administrar lo dicho anteriormente se utiliza la opción *high* para alta performance, *low* para el caso contrario, una opción *custom* para usuarios avanzados y *medium* para lograr un equilibrio entre rendimiento y uso de memoria. Esta última es la opción que viene configurada de forma predeterminada.

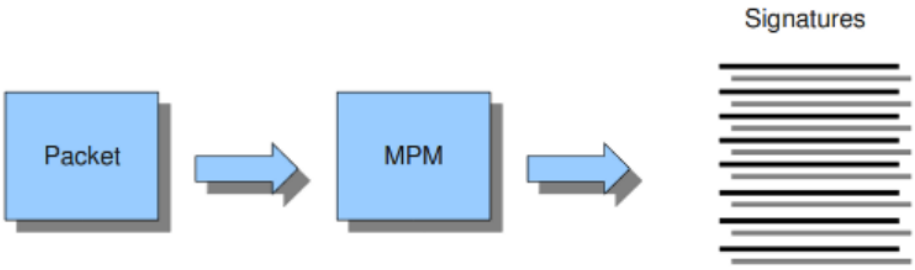
```
detect:
  profile: medium
  custom-values:
    toclient-groups: 2
    toserver-groups: 25
  sgh-mpm-context: auto
  inspection-recursion-limit: 3000
```

  Para configurar la propiedad de *multiple pattern matching* dirigirse al siguiente enlace.

- Motores de pre filtrado: surgieron debido a la gran cantidad de reglas disponibles a ser inspeccionadas de forma individual. Su función es la de tomar una condición de cada regla y agregarla al prefiltro para luego ser checkeada en un paso. El ejemplo más común es el *multiple pattern matching* o MPM, el cual toma un único patrón por regla y lo agrega al MPM. El ejemplo anterior solo será aplicado a todas aquellas reglas que poseen como mínimo una coincidencia de patrones en la fase MPM.
- Configuración de coincidencia de patrones (pattern matching): el multi-pattern-matcher forma parte del motor de detección de Suricata y su función es la de buscar múltiples patrones en una vez. Es decir que se utiliza un patrón de cada firma. De este modo,

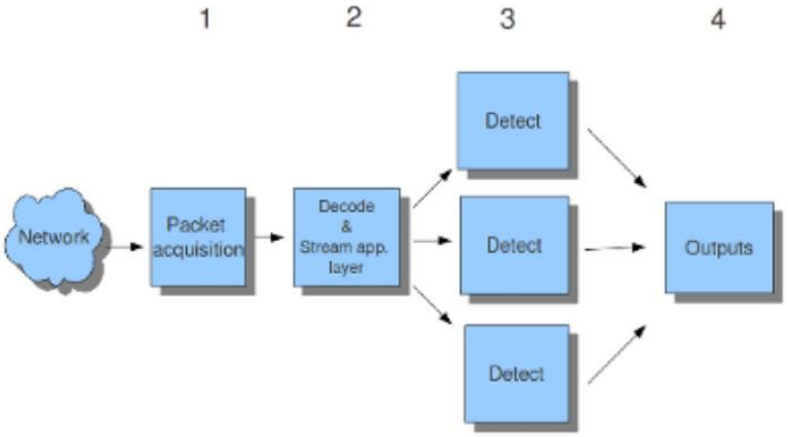
Suricata puede evitar de examinar muchas firmas teniendo que cuenta que cada una de estas solo puede coincidir cuando todos sus patrones lo hacen.
El procedimiento es el siguiente:

1. Ingresa un paquete.
2. El paquete es analizado con el multi-pattern-matcher para buscar patrones que coincidan.
3. Todos los patrones que coincidan serán procesados por Suricata (firmas).



Suricata posee varias implementaciones de algoritmos MPM. Para establecer el mismo se debe ingresar `mpm-algo: b2gc`
Luego de *mpm-algo* se puede optar por los siguientes algoritmos: *b2g*, *b2gc*, *b2gm*, *b3g*, *wumanber*, *ac* y *ac-gfbs*.
Para más información referida a la configuración ver *mpm-algo* y *mpm-context*.

- Threading: Suricata tiene la propiedad de ser multi-hilado, esto quiere decir que utiliza múltiples CPUs/núcleos CPU para procesar muchos paquetes de red simultáneamente. A diferencia de un motor de un solo núcleo en el que los paquetes son procesados de a uno por vez.
  Se distinguen cuatro módulos hilados:
  1. Adquisición del paquete: los paquetes provenientes de la red son leídos.
  2. Decodificación y transmisión en la capa de aplicación: se realiza la decodificación del paquete, el seguimiento del flujo y el re ensamblado del paquete.
  3. Detección: varios hilos operan de forma simultánea para realizar la comparativa con las firmas.
  4. Salida: se procesan todos los eventos y alertas.



Es importante mencionar que es el sistema operativo es el que determina qué núcleo trabaja con qué hilo. Cuando un núcleo está ocupado, se designará a otro núcleo para que

trabaje en el hilo de ejecución. Desde Suricata se puede establecer una especie de afinidad para que los hilos siempre trabajen con los mismos núcleos. Esto se realiza con el comando `set-cpu-affinity: yes`

- IP Defrag: Es común que los paquetes de red ingresen de forma fragmentada, esto implica que Suricata deba reconstruirlos para inspeccionarlos de forma precisa. Dicha tarea se realiza por un componente de Suricata llamado *motor de desfragmentación*.
  La desfragmentación posee tres opciones a configurar con respecto a los fragmentos. Estas son explicadas a continuación junto con sus valores por defecto:
  - *max-frags (65535)*, que indica la cantidad máxima de fragmentos que un paquete puede tener.
  - *prealloc (yes)*, indica que Suricata almacenará los fragmentos en memoria hasta encontrar a los demás.
  - *timeout (60)*, tiempo límite para recolectar todos los fragmentos de un paquete, pasado dicho tiempo se descartan.
- Ajuste de flujos (Flow Settings): en Suricata los flujos tienen una gran responsabilidad en el modo que se organizan internamente los datos. Un flujo es similar a una conexión, aunque más general. Todos los paquetes que poseen la misma tupla (protocolo, IP origen, IP destino, puerto origen, puerto destino) pertenecen al mismo flujo. Los paquetes que pertenecen a un flujo están conectados a él internamente. Es importante destacar que mantener la ubicación de todos los flujos redundará en una mayor utilización de memoria, para controlar esto se utilizan *timeouts*.
  Por otra parte, para mantener la ubicación de conexiones TCP se utiliza un motor de flujo que está compuesto por una parte que realiza el seguimiento del flujo monitoreando el estado de la conexión y, otra que reconstruye la corriente como era anteriormente para que sea reconocida por Suricata.
- Motor de salida: este motor contiene un subsistema de registro que muestra toda la salida excepto eventos y alertas. Además, brinda información en tiempo de ejecución sobre lo que está realizando el motor. Dicha información puede mostrarse mientras que el motor se inicia, en tiempo de ejecución o mientras el motor se apaga. El subsistema puede configurarse en modo *error*, *warning*, *informational* y *debug* en base a la cantidad de información queramos visualizar. `logging: default-log-level: info`
- NFQ: con la utilización de NFQUEUE en las reglas de *iptables*[8] se enviarán los paquetes a Suricata. Se distinguen diferentes modos:
  - Si el modo está configurado en *accept*, el paquete no será inspeccionado por el resto de las *iptables* luego de ser procesado por Suricata.
  - Si es *repeat* los paquetes serán insertados nuevamente a la primera regla de *iptables* luego de ser procesado por Suricata. Se utilizan marcas para evitar ciclos.
  - Si es *route* podremos enviar el paquete para ser analizado por otra herramienta luego ser haber sido procesado por Suricata.
- Reglas: los archivos que contienen las reglas que utiliza Suricata están configurados en base a categorías de riesgo. En base a esto se puede configurar Suricata para que utilice

[8] *Iptables* es una herramienta de cortafuegos que permite no solamente filtrar paquetes, sino también realizar traducción de direcciones de red (NAT) para IPv4 o mantener registros de log.

solo determinadas reglas, como por ejemplo cuando tenemos nuestro propio archivo de reglas.

Suricata posee un archivo de clasificación de reglas para proveer información sobre el propósito de cada regla.

Además, existe la posibilidad de definir variables a aplicar en las reglas para así establecer para qué dirección IP se aplica una determinada regla y con ello se ayuda a que sólo sean utilizadas las reglas relevantes.

- Capas de aplicación: los parsers SSL/TLS rastrean sesiones cifradas SSLv2, TLSv1, TLSv1.1 y TLSv1.2.

  El protocolo de detección se lleva a cabo a través de la utilización de patrones y realizando el sondeo en el puerto TCP/443 de manera predeterminada. La detección basada en patrones es independiente del puerto.

```
tls:
  enabled: yes
  detection-ports:
    dp: 443

  # Completely stop processing TLS/SSL session after the handshake
  # completed. If bypass is enabled this will also trigger flow
  # bypass. If disabled (the default), TLS/SSL session is still
  # tracked for Heartbleed and other anomalies.
  #no-reassemble: yes
```

  A causa de que no se realiza el descifrado del tráfico, el rastreo de la sesión tiene una utilidad limitada una vez que se completa el handshake. La opción *no-reassemble* mostrada arriba se establece en *false* de forma predeterminada y, en el mencionado estado no se realiza el descifrado del tráfico. Si dicha opción se establece en *true* todo el procesamiento de la sesión será frenado. La habilitación de *bypass* tiene como consecuencia que el flujo será bypaseado o saltado tanto por Suricata como por el método de captura.

3.7: Uso en modo “Prevención de Intrusos”

Suricata también tiene la capacidad de funcionar como un sistema preventivo de intrusos trabajando en la capa 3 (capa de red). Para esto, se debe configurar *iptables* para dicho propósito.

El primer paso es el de compilar Suricata con el soporte NFQ. Verificamos que tenemos NFQ habilitado en Suricata.

```
ale@ubuntu:~$ suricata --build-info
```

```
Suricata Configuration:
  AF_PACKET support:                       yes
  PF_RING support:                         no
  NFQueue support:                         yes
```

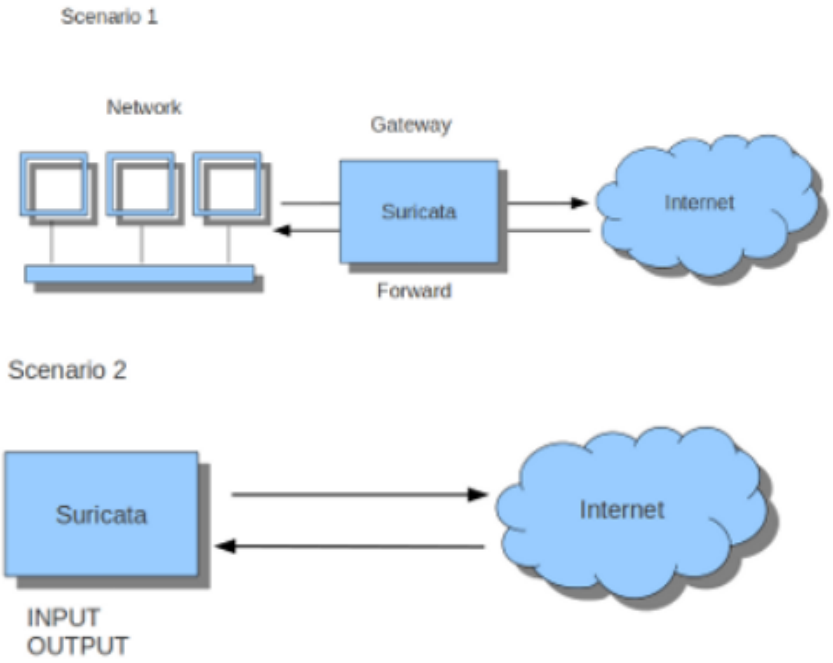
Para ejecutar Suricata en el modo NFQ se utiliza la opción *–q*. Dicho número le indica a Suricata la cola a utilizar.

```
ale@ubuntu:~$ sudo suricata -c /etc/suricata/suricata.yaml -q 0
13/7/2017 -- 20:30:08 - <Notice> - This is Suricata version 3.2.1 RELEASE
13/7/2017 -- 20:30:30 - <Notice> - all 6 packet processing threads, 4 management threads initialized, engine started.
^C13/7/2017 -- 20:30:35 - <Notice> - Signal Received.  Stopping engine.
13/7/2017 -- 20:30:36 - <Notice> - (RX-Q0) Treated: Pkts 0, Bytes 0, Errors 0
13/7/2017 -- 20:30:36 - <Notice> - (RX-Q0) Verdict: Accepted 0, Dropped 0, Replaced 0
```

Es importante mencionar que una vez que se frena la ejecución de Suricata se muestran una serie de estadísticas sobre las acciones que se tomaron para con los paquetes analizados.

Configuración de iptables

Como primera medida es importante conocer qué tráfico queremos que sea enviado a Suricata. Si el tráfico que atraviesa la computadora o el generado por la misma.



Si Suricata se ejecuta en una puerta de enlace o *Gateway* tiene como objetivo proteger las computadoras detrás de dicho *Gateway*. En este caso estamos tratando con el primer escenario: *forward_ing*. Si, por el contrario, Suricata realiza la protección de la computadora en la que está corriendo estaremos tratando con el segundo escenario: *host*. Es importante mencionar que los modos mencionados con anterioridad pueden combinarse.

La regla más sencilla en el escenario de Gateway es en la que se envía todo el tráfico a Suricata.

```
ale@ubuntu:~$ sudo iptables -I FORWARD -j NFQUEUE
```

En el caso del *host* las reglas más sencillas serían las de analizar todo el tráfico entrante y saliente del sistema.

```
ale@ubuntu:~$ sudo iptables -I OUTPUT -j NFQUEUE

ale@ubuntu:~$ sudo iptables -I INPUT -j NFQUEUE
```

A partir de dichas reglas Suricata tomará la información entrante y saliente, y la analizará utilizando su base de firmas.

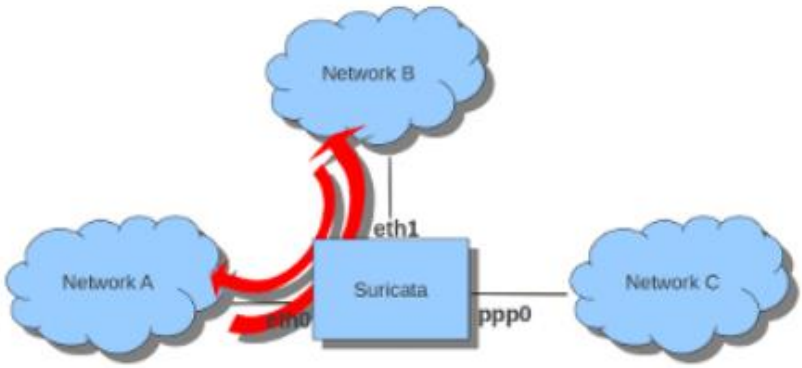
También es posible establecer el número de cola, si no lo hacemos se toma la cola 0 de forma predeterminada. Además, puerto fuente y destino con `--sport` y `–dport` respectivamente.

Si queremos verificar nuestras reglas de *iptables* mientras Suricata se está ejecutando:

```
ale@ubuntu:~$ sudo iptables -vnL
```

La descripción anterior de uso de *iptables* está orientada a IPv4. Para aplicarlo en IPv6 todos los comandos deben empezar con *ip6tables*.

Además, se puede utilizar *iptables* en múltiples redes. Solo se debe especificar la interfaz de red en cada regla con *–i*.



Si por algún motivo se necesita detener la ejecución de Suricata y utilizar internet se deben eliminar todas las reglas de *iptables* con el comando `sudo iptables –F`.

3.8: Archivos de salida

Eve (Extensible Event Format)

El formato Eve JSON se está convirtiendo en un estándar a la hora de exportar alertas y eventos, y se utiliza como salida de alertas y eventos (*HTTP*, *DNS*, *TLS*) en cualquier IDS. Otorga una gran flexibilidad al permitir un fácil post procesamiento por herramientas de terceros como pueden ser *Elasticsearch* o *DOM*.

Es un único archivo con formato JSON que se utiliza como salida de alertas y eventos (http, dns, tls). Permite que luego sea procesado fácilmente por herramientas de terceros.

Ejemplo de información contenida en una línea:

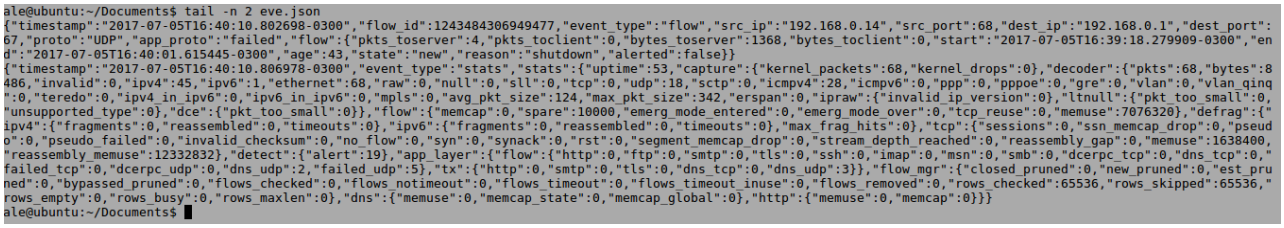
```
{"timestamp":"2017-07-05T16:39:40.866685-0300",
"flow_id":514967660673405,
"in_iface":"ens38",
"event_type":"dns",
```

```
"src_ip":"192.168.0.14",
"src_port":32825,
"dest_ip":"190.1.0.196",
"dest_port":53,
"proto":"UDP",
"dns":{
    "type":"query","id":40488,"rrname":"ntp.ubuntu.com","rrtype":"A","tx_id":0
      }
}
```


Como podemos ver en la imagen anterior, todos los ingresos que posee el archivo JSON comparten una misma estructura y el campo *Event type* que indica el tipo de registro. Este último se clasifica en:

- Alert: posee un campo *action* que puede tomar dos valores, *allowed* (permitido) ó *blocked* (bloqueado). Se establece en *allowed* a menos que la regla haya utilizado la acción *drop* (soltar) y Suricata se encuentre en modo IPS, o cuando la regla usó la acción *reject* (rechazar). Ejemplo: "`action":"allowed`"
- HTTP: posee varios campos, entre ellos el *hostname* que indica a quien pertenece el evento HTTP, el *url* del hostname que fue accedido, el *http_user_agent* que denota la aplicación usada para acceder al contenido HTTP, el *http_content_type* que indica el tipo de dato devuelto y una *cookie*. Además, se pueden añadir campos si se utiliza el registro extendido como pueden ser la longitud del cuerpo HTTP, el protocolo, el método HTTP (GET, POST, etc.), entre otros.
- DNS: contiene diferentes campos que representan los diferentes tipos de eventos DNS. Entre estos se encuentran el campo *type* que puede ser *query* o *answer*, el *id*, el *ttl*, etc.

Al momento de poder observar la salida del archivo Eve, es importante mencionar que, si bien la salida es fácilmente entendible por cualquier persona gracias a los beneficios de JSON, la gran cantidad de datos que puede contener cada evento hace prácticamente imposible que una persona pueda hacerlo sin ningún tipo de herramienta adicional. La siguiente captura nos da una idea de lo mencionado:



```
ale@ubuntu:~/Documents$ tail -n 2 eve.json
{"timestamp":"2017-07-05T16:40:10.802698-0300","flow_id":1243484306949477,"event_type":"flow","src_ip":"192.168.0.14","src_port":68,"dest_ip":"192.168.0.1","dest_port":
67,"proto":"UDP","app_proto":"failed","flow":{"pkts_toserver":4,"pkts_toclient":0,"bytes_toserver":1368,"bytes_toclient":0,"start":"2017-07-05T16:39:18.279909-0300","en
d":"2017-07-05T16:40:01.615445-0300","age":43,"state":"new","reason":"shutdown","alerted":false}}
{"timestamp":"2017-07-05T16:40:10.806978-0300","event_type":"stats","stats":{"uptime":53,"capture":{"kernel_packets":68,"kernel_drops":0},"decoder":{"pkts":68,"bytes":8
486,"invalid":0,"ipv4":45,"ipv6":1,"ethernet":68,"raw":0,"null":0,"sll":0,"tcp":0,"udp":18,"sctp":0,"icmpv4":28,"icmpv6":0,"ppp":0,"pppoe":0,"gre":0,"vlan":0,"vlan_qinq
":0,"teredo":0,"ipv4_in_ipv6":0,"ipv6_in_ipv6":0,"mpls":0,"avg_pkt_size":124,"max_pkt_size":342,"erspan":0,"ipraw":{"invalid_ip_version":0},"ltnull":{"pkt_too_small":0,
"unsupported_type":0},"dce":{"pkt_too_small":0}},"flow":{"memcap":0,"spare":10000,"emerg_mode_entered":0,"emerg_mode_over":0,"tcp_reuse":0,"memuse":7076320},"defrag":{"
ipv4":{"fragments":0,"reassembled":0,"timeouts":0},"ipv6":{"fragments":0,"reassembled":0,"timeouts":0},"max_frag_hits":0},"tcp":{"sessions":0,"ssn_memcap_drop":0,"pseud
o":0,"pseudo_failed":0,"invalid_checksum":0,"no_flow":0,"syn":0,"synack":0,"rst":0,"segment_memcap_drop":0,"stream_depth_reached":0,"reassembly_gap":0,"memuse":1638400,
"reassembly_memuse":12332832},"detect":{"alert":19},"app_layer":{"flow":{"http":0,"ftp":0,"smtp":0,"tls":0,"ssh":0,"imap":0,"msn":0,"smb":0,"dcerpc_tcp":0,"dns_tcp":0,"
failed_tcp":0,"dcerpc_udp":0,"dns_udp":2,"failed_udp":5},"tx":{"http":0,"smtp":0,"tls":0,"dns_tcp":0,"dns_udp":3}},"flow_mgr":{"closed_pruned":0,"new_pruned":0,"est_pru
ned":0,"bypassed_pruned":0,"flows_checked":0,"flows_notimeout":0,"flows_timeout":0,"flows_timeout_inuse":0,"flows_removed":0,"rows_checked":65536,"rows_skipped":65536,"
rows_empty":0,"rows_busy":0,"rows_maxlen":0},"dns":{"memuse":0,"memcap_state":0,"memcap_global":0},"http":{"memuse":0,"memcap":0}}}
ale@ubuntu:~/Documents$
```


La utilización de herramientas estándar de Unix como *grep* no nos brinda una muy buena solución ya que permite mostrar solo la porción de las estadísticas que nos interesan. Además, es importante destacar que los campos de JSON no se encuentran ordenados. A partir de lo mencionado se introduce la herramienta *jq*, la cual tiene como funcionalidad el parseo de un archivo con formato JSON.

Instalamos dicha herramienta:

```
ale@ubuntu:~$ sudo apt-get install jq
```

Luego, para utilizarlo podemos colorear la salida con el siguiente comando.

```
ale@ubuntu:/var/log/suricata$ sudo tail -n1 eve.json | jq '.'
```

Donde *n* indica el número de líneas a mostrar, si *n=1* se mostrará la última.

Y parte de la salida será.


```
{
  "timestamp": "2017-07-05T16:40:10.806978-0300",
  "event_type": "stats",
  "stats": {
    "uptime": 53,
    "capture": {
      "kernel_packets": 68,
      "kernel_drops": 0
    },
    "decoder": {
      "pkts": 68,
      "bytes": 8486,
      "invalid": 0,
      "ipv4": 45,
      "ipv6": 1,
      "ethernet": 68,
      "raw": 0,
      "null": 0,
      "sll": 0,
      "tcp": 0,
      "udp": 18,
      "sctp": 0,
      "icmpv4": 28,
      "icmpv6": 0,
      "ppp": 0,
      "pppoe": 0,
      "gre": 0,
      "vlan": 0,
      "vlan_qinq": 0,
      "teredo": 0,
      "ipv4_in_ipv6": 0,
      "ipv6_in_ipv6": 0,
      "mpls": 0,
      "avg_pkt_size": 124,
      "max_pkt_size": 342,
      "erspan": 0,
      "ipraw": {
        "invalid_ip_version": 0
      },
      "ltnull": {
        "pkt_too_small": 0,
        "unsupported_type": 0
      },
      "dce": {
        "pkt_too_small": 0
```


También, se puede mostrar la salida de un evento por línea añadiendo el flag *–c*.

```
ale@ubuntu:/var/log/suricata$ sudo tail -n1 eve.json | jq '.' -c
```

Y parte de la salida será.


```
{"timestamp":"2017-07-05T16:40:10.806978-0300","event_type":"stats","stats":{"uptime":53,"capture":{"kernel_packe
6,"invalid":0,"ipv4":45,"ipv6":1,"ethernet":68,"raw":0,"null":0,"sll":0,"tcp":0,"udp":18,"sctp":0,"icmpv4":28,"ic
"teredo":0,"ipv4_in_ipv6":0,"ipv6_in_ipv6":0,"mpls":0,"avg_pkt_size":124,"max_pkt_size":342,"erspan":0,"ipraw":{"
ported_type":0},"dce":{"pkt_too_small":0}},"flow":{"memcap":0,"spare":10000,"emerg_mode_entered":0,"emerg_mode_ov
fragments":0,"reassembled":0,"timeouts":0},"ipv6":{"fragments":0,"reassembled":0,"timeouts":0},"max_frag_hits":0},
do_failed":0,"invalid_checksum":0,"no_flow":0,"syn":0,"synack":0,"rst":0,"segment_memcap_drop":0,"stream_depth_re
memuse":12332832},"detect":{"alert":19},"app_layer":{"flow":{"http":0,"ftp":0,"smtp":0,"tls":0,"ssh":0,"imap":0,"
"dcerpc_udp":0,"dns_udp":2,"failed_udp":5},"tx":{"http":0,"smtp":0,"tls":0,"dns_tcp":0,"dns_udp":3}},"flow_mgr":{"
_pruned":0,"flows_checked":0,"flows_notimeout":0,"flows_timeout":0,"flows_timeout_inuse":0,"flows_removed":0,"rows
s_busy":0,"rows_maxlen":0},"dns":{"memuse":0,"memcap_state":0,"memcap_global":0},"http":{"memuse":0,"memcap":0}}}
```


Suricata.log

Archivo de registro que muestra advertencias (*warnings*), errores e información cuando se ejecuta y cuando finaliza Suricata.

```
11/7/2017 -- 15:58:29 - <Notice> - This is Suricata version 3.2.1 RELEASE
11/7/2017 -- 15:58:49 - <Notice> - all 4 packet processing threads, 4 management threads initialized, engine started.
11/7/2017 -- 15:59:12 - <Notice> - Signal Received.  Stopping engine.
11/7/2017 -- 15:59:12 - <Notice> - Stats for 'ens38':  pkts: 34, drop: 0 (0.00%), invalid chksum: 0
11/7/2017 -- 18:23:30 - <Notice> - This is Suricata version 3.2.1 RELEASE
11/7/2017 -- 18:24:02 - <Notice> - all 4 packet processing threads, 4 management threads initialized, engine started.
11/7/2017 -- 18:24:22 - <Notice> - Signal Received.  Stopping engine.
11/7/2017 -- 18:24:23 - <Notice> - Stats for 'ens38':  pkts: 9, drop: 0 (0.00%), invalid chksum: 0
11/7/2017 -- 18:27:11 - <Notice> - This is Suricata version 3.2.1 RELEASE
11/7/2017 -- 18:27:31 - <Notice> - all 4 packet processing threads, 4 management threads initialized, engine started.
11/7/2017 -- 18:28:08 - <Notice> - Signal Received.  Stopping engine.
11/7/2017 -- 18:28:31 - <Notice> - Stats for 'ens38':  pkts: 1012917, drop: 30837 (3.04%), invalid chksum: 0
```

Suricata-start.log

Almacena información sobre problemas que pueden surgir al ejecutar Suricata.

Fast.log

Archivo que contiene información sintetizada en una línea sobre los sucesos ocurridos durante la ejecución de Suricata. Ejemplo:

```
07/11/2017-19:56:35.961708  [**] [1:2210045:2] SURICATA STREAM Packet with invalid ack
[**] [Classification: Generic Protocol Command Decode]
[Priority: 3] {TCP} 192.168.0.104:80 -> 192.168.0.105:58338
07/11/2017-19:56:35.961708  [**] [1:2210046:2] SURICATA STREAM SHUTDOWN RST invalid ack
[**] [Classification: Generic Protocol Command Decode]
[Priority: 3] {TCP} 192.168.0.104:80 -> 192.168.0.105:58338
07/11/2017-19:56:40.269125  [**] [1:2022973:1] ET POLICY Possible Kali Linux hostname in DHCP Request Packet
[**] [Classification: Potential Corporate Privacy Violation]
[Priority: 1] {UDP} 192.168.0.105:68 -> 192.168.0.1:67
```

Stats.log

Contiene estadísticas que son creadas cada una cantidad de tiempo predefinida en el archivo de configuración de Suricata. Ejemplo:

```
Date: 7/11/2017 -- 19:56:40 (uptime: 0d, 00h 00m 30s)
----------------------------------------------------------------------------------
Counter                                     | TM Name                   | Value
----------------------------------------------------------------------------------
capture.kernel_packets                      | Total                     | 383873
capture.kernel_drops                        | Total                     | 355140
decoder.pkts                                | Total                     | 28823
decoder.bytes                               | Total                     | 1631765
decoder.ipv4                                | Total                     | 28818
decoder.ethernet                            | Total                     | 28823
decoder.tcp                                 | Total                     | 28811
decoder.udp                                 | Total                     | 7
decoder.avg_pkt_size                        | Total                     | 56
decoder.max_pkt_size                        | Total                     | 342
tcp.sessions                                | Total                     | 12029
tcp.syn                                     | Total                     | 12242
tcp.rst                                     | Total                     | 16569
detect.alert                                | Total                     | 5334
app_layer.flow.failed_udp                   | Total                     | 3
flow.spare                                  | Total                     | 10000
flow_mgr.rows_checked                       | Total                     | 65536
flow_mgr.rows_skipped                       | Total                     | 65536
tcp.memuse                                  | Total                     | 2405800
tcp.reassembly_memuse                       | Total                     | 12332832
flow.memuse                                 | Total                     | 10582144
----------------------------------------------------------------------------------
```

3.9: Experimentación con diversos casos de prueba

En esta sección se realizarán una serie de pruebas a modo de comprobar la capacidad de respuesta y la flexibilidad que posee Suricata tanto en el modo prevención como en el de detección de intrusos.

Se utiliza una máquina virtual adicional que posee una distribución basada en *Debian GNU/Linux* llamada *Kali Linux* en su versión *2017.1*. Este es un sistema operativo diseñado principalmente para la auditoria y seguridad informática en general. Estará ubicado en la misma red LAN y posibilitará la realización de la mayoría de las pruebas. Para permitir que ambas máquinas virtuales estén en la misma LAN se deben configurar sus placas de red en modo *Bridge* desde la configuración de VMware.

La realización de los siguientes test va a estar basada en un conjunto de reglas personalizadas. Para añadirlas creamos un archivo propio de reglas al que llamamos *local.rules* y lo ubicamos en el mismo lugar donde se encuentran los demás archivos de reglas, esto es, en: */etc/suricata/rules*.

```
ale@ubuntu:/etc/suricata/rules$ ls
botcc.portgrouped.rules         emerging-icmp_info.rules       emerging-user_agents.rules
botcc.rules                     emerging-icmp.rules            emerging-voip.rules
BSD-License.txt                 emerging-imap.rules            emerging-web_client.rules
ciarmy.rules                    emerging-inappropriate.rules   emerging-web_server.rules
classification.config           emerging-info.rules            emerging-web_specific_apps.rules
compromised-ips.txt             emerging-malware.rules         emerging-worm.rules
compromised.rules               emerging-misc.rules            files.rules
decoder-events.rules            emerging-mobile_malware.rules  gen-msg.map
dns-events.rules                emerging-netbios.rules         gpl-2.0.txt
drop.rules                      emerging-p2p.rules             http-events.rules
dshield.rules                   emerging-policy.rules          local.rules
emerging-activex.rules          emerging-pop3.rules            rbn-malvertisers.rules
emerging-attack_response.rules  emerging-rpc.rules             rbn.rules
emerging-chat.rules             emerging-scada.rules           reference.config
emerging.conf                   emerging-scan.rules            sid-msg.map
emerging-current_events.rules   emerging-shellcode.rules       smtp-events.rules
emerging-deleted.rules          emerging-smtp.rules            stream-events.rules
emerging-dns.rules              emerging-snmp.rules            suricata-1.3-open.txt
emerging-dos.rules              emerging-sql.rules             threshold.config
emerging-exploit.rules          emerging-telnet.rules          tls-events.rules
emerging-ftp.rules              emerging-tftp.rules            tor.rules
emerging-games.rules            emerging-trojan.rules          unicode.map
```

Además, debemos crear una clase que agrupa las reglas propias y le asignamos una prioridad predeterminada que luego puede ser sobrescrita en cada regla. Esto se realiza en el archivo *classification.config* ubicado en la misma carpeta que las reglas.

La creación de la clase se hará en cada prueba ya que en cada una se crearán reglas que no tienen relación entre sí.

Por último, incluimos el archivo de reglas en el archivo de configuración *suricata.yaml* para que pueda ser tenido en cuenta.

```
##
## Step 2: select the rules to enable or disable
##

default-rule-path: /etc/suricata/rules
rule-files:

 - local.rules #reglaPropia

 - emerging-deleted.rules #n
 - files.rules #n

 - botcc.rules
 - botcc.portgrouped.rules #d
 - ciarmy.rules
 - compromised.rules
 - drop.rules
 - dshield.rules
 - emerging-activex.rules #d
 - emerging-attack_response.rules
```

3.9.1: Ping

Sistema de detección de intrusos (IDS)

La utilidad *ping* opera en la capa de red del protocolo TCP/IP. Es usada para testear el alcance de un host en una red IP a través del envío de paquetes ICMP (solicitud, *request*; respuesta, *reply*). A partir de esto, realiza una medición del tiempo que tardó el paquete en ir sumado al que tardó en volver.

La prueba realizada consiste en añadir la clase que contiene nuestra regla en el archivo *classification.config*, una descripción y una prioridad que, en este caso, será baja ya que solo se trata de un Ping.

```
#
# config classification:shortname,short description,priority
#

#Clasificacion de mis reglas
config classification: regla-ping,Ping recibido,4

#Traditional classifications. These will be replaced soon

config classification: not-suspicious,Not Suspicious Traffic,3
config classification: unknown,Unknown Traffic,3
config classification: bad-unknown,Potentially Bad Traffic, 2
```

Creamos la siguiente regla en nuestro archivo *local.rules*.

```
alert icmp any any -> $HOME_NET any
(msg:"Alguien esta haciendo ping";
itype:8;
classtype:regla-ping;
sid:1000000;)
```

En ella especificamos la acción, el protocolo a utilizar, la IP y el puerto origen, la IP y el puerto destino, un mensaje para mostrar en el registro *eve.json*, el *itype* 8 que denota que detectará un mensaje del tipo *ICMP echo request*, la clase que agrupa dicha regla y un número de identificación de la misma.

Una vez identificada la IP que posee el host con contiene a Suricata ejecutamos el IDS.

```
ale@ubuntu:~$ sudo LD_PRELOAD="/usr/lib/libtcmalloc_minimal.so.4" suricata -c /etc/suricata/suricata.yaml -i ens33
[sudo] password for ale:
2/12/2017 -- 19:52:50 - <Notice> - This is Suricata version 3.2.1 RELEASE
2/12/2017 -- 19:53:28 - <Notice> - all 4 packet processing threads, 4 management threads initialized, engine started.
```

Realizamos el ping desde el *Kali Linux*.

```
root@kali:~# ping 192.168.0.104
PING 192.168.0.104 (192.168.0.104) 56(84) bytes of data.
64 bytes from 192.168.0.104: icmp_seq=1 ttl=64 time=0.546 ms
64 bytes from 192.168.0.104: icmp_seq=2 ttl=64 time=0.646 ms
64 bytes from 192.168.0.104: icmp_seq=3 ttl=64 time=0.691 ms
64 bytes from 192.168.0.104: icmp_seq=4 ttl=64 time=0.347 ms
64 bytes from 192.168.0.104: icmp_seq=5 ttl=64 time=0.683 ms
64 bytes from 192.168.0.104: icmp_seq=6 ttl=64 time=0.676 ms
64 bytes from 192.168.0.104: icmp_seq=7 ttl=64 time=0.698 ms
^C
--- 192.168.0.104 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6117ms
rtt min/avg/max/mdev = 0.347/0.612/0.698/0.120 ms
root@kali:~#
```

Luego procedo a analizar el archivo de registro *eve.json* y se identifican varias entradas similares a la que se muestra a continuación.


```
{
  "timestamp": "2017-07-11T15:24:21.204868-0300",
  "in_iface": "ens38",
  "event_type": "alert",
  "src_ip": "192.168.0.105",
  "dest_ip": "192.168.0.104",
  "proto": "ICMP",
  "icmp_type": 8,
  "icmp_code": 0,
  "alert": {
    "action": "allowed",
    "gid": 1,
    "signature_id": 1000000,
    "rev": 0,
    "signature": "Alguien esta haciendo ping",
    "category": "Ping recibido",
    "severity": 4
  }
```


Se pueden observar los tipos y códigos de ICMP donde el tipo 8 denota un *echo* y el código 0 se utiliza para emparejar el *echo* con la respuesta (RFC 792). Además, se puede observar información correspondiente a la clase a la que pertenece. Es importante destacar que como la acción es *alert* se permite el ping (*allowed*) pero se generó una alerta como el mostrado.

También podemos ver información de lo ocurrido sintetizada en una línea en el archivo *fast.log*.

```
07/11/2017-15:24:16.111608  [**] [1:1000000:0] Alguien esta haciendo ping
[**] [Classification: Ping recibido] [Priority: 4]
{ICMP} 192.168.0.105:8 -> 192.168.0.104:0
```

En base a lo observado anteriormente, modificando la acción de la regla a *reject* se puede observar lo siguiente:

```
reject icmp any any -> $HOME_NET any
(msg:"Alguien esta haciendo ping";
itype:8;
classtype:regla-ping;
sid:1000000;)
```

Realizamos el ping.

```
root@kali:~# ping 192.168.0.104
PING 192.168.0.104 (192.168.0.104) 56(84) bytes of data.
64 bytes from 192.168.0.104: icmp_seq=1 ttl=64 time=0.293 ms
From 192.168.0.104 icmp_seq=1 Destination Host Prohibited
64 bytes from 192.168.0.104: icmp_seq=2 ttl=64 time=0.702 ms
From 192.168.0.104 icmp_seq=2 Destination Host Prohibited
64 bytes from 192.168.0.104: icmp_seq=3 ttl=64 time=0.715 ms
From 192.168.0.104 icmp_seq=3 Destination Host Prohibited
64 bytes from 192.168.0.104: icmp_seq=4 ttl=64 time=0.591 ms
From 192.168.0.104 icmp_seq=4 Destination Host Prohibited
^C
--- 192.168.0.104 ping statistics ---
4 packets transmitted, 4 received, +4 errors, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 0.293/0.575/0.715/0.170 ms
root@kali:~#
```

Y como se puede observar, el IDS nos rechaza el paquete de manera activa.

El archivo de registro de Suricata muestra lo siguiente:


```
{
  "timestamp": "2017-07-11T15:59:06.719255-0300",
  "in_iface": "ens38",
  "event_type": "alert",
  "src_ip": "192.168.0.105",
  "dest_ip": "192.168.0.104",
  "proto": "ICMP",
  "icmp_type": 8,
  "icmp_code": 0,
  "alert": {
    "action": "blocked",
    "gid": 1,
    "signature_id": 1000000,
    "rev": 0,
    "signature": "Alguien esta haciendo ping",
    "category": "Ping recibido",
    "severity": 4
  }
```


En donde se generó una alerta que especifica la acción tomada, que en este caso es la de bloquear (*blocked*) el ping.

En este caso el archivo *fast.log* también muestra lo ocurrido, donde *wDrop* (*Would Drop*) evidencia que la acción utilizada es *reject*.

```
07/11/2017-15:59:03.713293  [wDrop] [**] [1:1000000:0] Alguien esta haciendo ping
[**][Classification: Ping recibido] [Priority: 4]
{ICMP} 192.168.0.105:8 -> 192.168.0.104:0
```

Sistema de prevención de intrusos (IPS)

Cuando se configura Suricata como sistema de prevención de intrusos en modo *host* lo primero que se debe hacer es añadir reglas a *iptables* para especificar que tráfico queremos que Suricata analice. La manera más sencilla es la de configurar Suricata para que analice todo el tráfico como se explicó en la sección *iptables*. Sin embargo, en este caso solo queremos que sólo sea analizado el tráfico ICMP entrante generado por el comando, para esto agregamos la siguiente regla:

```
ale@ubuntu:~$ sudo iptables -I INPUT -p icmp  -j NFQUEUE
```

Es importante destacar que la regla será añadida cuando se ejecute Suricata IPS, por lo que si utilizamos el comando *iptables –vnL* para observar todas las reglas no será encontrada.

Ahora vamos a agregar una regla a nuestro archivo de reglas *local.rules* para que realice el *drop* de los paquetes ICMP.

```
drop icmp any any -> $HOME_NET any
(msg:"Alguien esta haciendo ping";
itype:8;
classtype:regla-ping;
sid:1000014;)
```

Ejecutamos Suricata en modo IPS:

```
ale@ubuntu:~$ sudo LD_PRELOAD="/usr/lib/libtcmalloc_minimal.so.4" suricata -c /etc/suricata/suricata.yaml -q 0
[sudo] password for ale:
14/7/2017 -- 15:41:50 - <Notice> - This is Suricata version 3.2.1 RELEASE
14/7/2017 -- 15:42:09 - <Notice> - all 6 packet processing threads, 4 management threads initialized, engine started.
```

Verificamos que se agregó la regla a *iptables*:

```
ale@ubuntu:~$ sudo iptables -vnL
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 NFQUEUE    icmp --  *      *       0.0.0.0/0            0.0.0.0/0           NFQUEUE num 0

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 3 packets, 984 bytes)
 pkts bytes target     prot opt in     out     source               destination
```

Ahora nos dirigimos a *Kali Linux* para ejecutar el comando *ping* hacia Suricata:

```
root@kali:~# ping 192.168.0.104
PING 192.168.0.104 (192.168.0.104) 56(84) bytes of data.
^C
--- 192.168.0.104 ping statistics ---
10 packets transmitted, 0 received, 100% packet loss, time 9211ms
```

Como se puede observar, los paquetes no llegan a destino por la acción de la regla, esto quiere decir que del lado de Kali Linux se interpreta que no existe una computadora con dicha IP ya que la respuesta no da ningún tipo de indicio. Por lo tanto, la presencia de Suricata será invisible ante este tipo de comando.

Al detener la ejecución de Suricata podemos ver una síntesis de las estadísticas durante la ejecución:

```
^C14/7/2017 -- 15:49:19 - <Notice> - Signal Received.  Stopping engine.
14/7/2017 -- 15:49:19 - <Notice> - (RX-Q0) Treated: Pkts 10, Bytes 840, Errors 0
14/7/2017 -- 15:49:19 - <Notice> - (RX-Q0) Verdict: Accepted 0, Dropped 10, Replaced 0
```

Aquí se muestra la cantidad de paquetes soltados en base a la cantidad total de paquetes recibidos.

Si bien el archivo *eve.json* muestra el mismo mensaje que en el caso de IDS con la acción *reject*, el archivo *fast.log* evidencia lo ocurrido mostrando *Drop* y no *wDrop* como en dicho caso:

```
07/14/2017-15:48:27.725580  [Drop] [**] [1:1000014:0] Alguien esta haciendo ping
[**] [Classification: Ping recibido] [Priority: 4]
{ICMP} 192.168.0.105:8 -> 192.168.0.104:0
```

Si se quiere que Suricata deje de soltar dichos paquetes ICMP no alcanza con detener su ejecución. Se debe quitar la regla de *local.rules* y volver a ejecutar Suricata en modo IPS para que note el cambio realizado.

Finalizando con esta prueba se realizó un test para corroborar la prioridad del orden de acción explicado con anterioridad. Para ello, se crearon cuatro reglas con todas las acciones posibles (*pass*, *drop*, *reject*, *alert*).

```
pass icmp any any -> $HOME_NET any (msg:"Alguien esta haciendo ping"; itype:8; classtype:regla-ping; sid:1000013;)
drop icmp any any -> $HOME_NET any (msg:"Alguien esta haciendo ping"; itype:8; classtype:regla-ping; sid:1000014;)
reject icmp any any -> $HOME_NET any (msg:"Alguien esta haciendo ping"; itype:8; classtype:regla-ping; sid:1000000;)
alert icmp any any -> $HOME_NET any (msg:"Alguien esta haciendo ping"; itype:8; classtype:regla-ping; sid:1000012;)
```

Al realizar el *ping –c* 4 para el envío de cuatro *echo request* desde el *Kali* se pudo observar que las solicitudes fueron aceptadas al actuar la acción *pass* por sobre las demás. Por lo tanto, no se encontró información en el archivo *fast.log*.

Si ahora repetimos la prueba colocando solo las reglas *drop*, *reject* y *alert*, se pueden visualizar dos rechazos activos de paquete (proveniente del *reject*) y en el *fast.log* se puede advertir que las reglas son utilizadas en el orden que se explicaron (ver los *sid* en la siguiente imagen).

```
[wDrop] [**] [1:1000014:0] Alguien esta haciendo ping [**] [Classification: Ping r
[wDrop] [**] [1:1000000:0] Alguien esta haciendo ping [**] [Classification: Ping r
[**] [1:1000012:0] Alguien esta haciendo ping [**] [Classification: Ping recibido]
```

3.9.2: DoS

Sistema de detección de intrusos (IDS)

Un ataque *DoS*, también llamado ataque de denegación de servicio, provoca que un servicio o recurso sea inaccesible a los usuarios legítimos. Puede estar dirigido a un grupo de computadoras o red y se generan como consecuencia de la saturación de los puertos con múltiples flujos de información, haciendo que el servidor se sobrecargue y no pueda seguir prestando su servicio.

Primero se crea la clase en el archivo *classification.config* que contendrá a la regla que detecte o prevenga ataques DoS junto con una prioridad alta debido a la importancia de este ataque.

```
#
# config classification:shortname,short description,priority
#

#Clasificacion de mis reglas
config classification: regla-DoS,Posible DoS,1
```

Luego, creamos la regla que alerte sobre el posible ataque.

```
alert tcp any any -> $HOME_NET $HTTP_PORTS
(flags: S;
msg:"Posible ataque TCP DoS!";
flow: to_server, stateless;
threshold: type both, track by_dst, count 70, seconds 10;
classtype:regla-DoS;
sid:1000009;)
```

Como se puede observar el protocolo es TCP, el origen puede ser cualquier IP y puerto, el destino tiene que ser alguna de las IPs que configuramos como locales y por cuestiones de performance se estableció que el puerto receptor sea solo el 80 (de lo contrario podría ser *any*).

flags:S: detecta los paquetes TCP que intentan sincronizar (*SYN*), es decir, que realizan un intento de conexión. Identificarlos permite que se realice alguna acción sobre los mismos.

Flow: to_server, stateless: indica que el match se realizara cuando se reciba una solicitud del cliente (Kali) al servidor (Ubuntu) y, tanto con paquetes que no son parte de una conexión establecida. Es decir, sería como tener en cuenta paquetes de modo *not_established*. El protocolo *stateless* trata a cada petición como una transacción independiente que no tiene relación con cualquier solicitud anterior.

Threshold: type both, track by_dst, count 70, seconds 10: dicho parámetro se utiliza para controlar la frecuencia de alertas de la regla. El tipo *both* combina el tipo *threshold* (establece un umbral a partir del cual generará una alerta), con el tipo *limit* (establece un límite al número de alertas generadas). El parámetro *track by_dst* especifica que se debe tener en cuenta hacia dónde va dirigida la conexión. En este caso, se generará una alerta si hay 70 o más paquetes TCP que cumplan las condiciones en 10 segundos.

Luego, ejecutamos el IDS y nos ubicamos en el *Kali Linux* para realizar el ataque DoS como se muestra en la siguiente imagen:

```
root@kali:~# hping3 --flood -p 80 -S 192.168.0.104 -I eth0
HPING 192.168.0.104 (eth0 192.168.0.104): S set, 40 headers + 0 data bytes
hping in flood mode, no replies will be shown
^C
--- 192.168.0.104 hping statistic ---
249281 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
root@kali:~#
```

El comando *hping3* se utiliza para enviar paquetes TCP a un *host* (en este caso, a nuestra máquina virtual con Suricata).

--flood: se encarga de enviar los paquetes lo más rápido posible sin tener en cuenta las respuestas a los mismos (asincrónicamente).

-p: especificamos el puerto al cual se enviarán los paquetes.

-*S:* serán enviados paquetes con el flag activo para sincronizar (*SYN*).

-*I:* especificamos la interfaz del emisor, es decir a través de la cual enviaremos los paquetes.

En cuestión de segundos el comando envía la cantidad de paquetes mostrada y luego se frena la ejecución del mismo.

Del lado de Suricata, al detener la ejecución del IDS podemos observar las siguientes estadísticas:

```
^C12/7/2017 -- 11:01:14 - <Notice> - Signal Received.  Stopping engine.
12/7/2017 -- 11:01:21 - <Notice> - Stats for 'ens38':  pkts: 498579, drop: 47214 (9.47%), invalid chksum: 0
```

Podemos interpretar que la gran cantidad de paquetes enviados hizo que las colas que posee Suricata para la recepción de los mismos se llenen y el IDS empiece a desechar paquetes (*drop*).

En el archivo de configuración *suricata.yaml* se puede modificar la cantidad máxima de paquetes que pueden estar pendientes, el valor actual es: *max-pending-packets: 1024*. Si a modo de ejemplo cambiamos el valor por 10240 y se vuelve a ejecutar la prueba anterior, la cantidad de paquetes soltados será mucho menor. Esto se puede observar en la siguiente imagen:

```
^C15/7/2017 -- 18:41:18 - <Notice> - Signal Received.  Stopping engine.
15/7/2017 -- 18:41:29 - <Notice> - Stats for 'ens38':  pkts: 538962, drop: 499 (0.09%), invalid chksum: 0
```

La configuración anterior nos asegura que cada CPU no este ocioso porque habrá mayor cantidad de paquetes por cada hilo de procesamiento, aunque tendrá un impacto negativo en la memoria ya que tendrá que almacenar una mayor cantidad de paquetes.

Al analizar el archivo de registro *eve.json* podemos identificar que no sólo se asoció el ataque a la regla creada, sino una perteneciente a Suricata que alerta sobre un paquete con *ack* invalido.


```
{
 "timestamp":"2017-07-12T11:01:02.116448-0300",
 "flow_id":1487764518054989,
 "in_iface":"ens38",
 "event_type":"alert",
 "src_ip":"192.168.0.104",
 "src_port":80,
 "dest_ip":"192.168.0.105",
 "dest_port":8191,
 "proto":"TCP",
 "alert":{
  "action":"allowed",
  "gid":1,
  "signature_id":2210045,
  "rev":2,
  "signature":"SURICATA STREAM Packet with invalid ack",
  "category":"Generic Protocol Command Decode",
  "severity":3
 }
}
```


El registro de nuestra regla:

```
{
 "timestamp":"2017-07-12T11:00:59.517417-0300",
 "flow_id":1230672070698281,
 "in_iface":"ens38",
 "event_type":"alert",
 "src_ip":"192.168.0.105",
 "src_port":1825,
 "dest_ip":"192.168.0.104",
 "dest_port":80,
 "proto":"TCP",
 "alert":{
  "action":"allowed",
  "gid":1,
  "signature_id":1000009,
  "rev":0,
  "signature":"Posible ataque TCP DoS!",
  "category":"Posible DoS",
  "severity":1
 }
}
```


Si modificamos la acción de nuestra regla por *reject* y mantenemos la cantidad máxima de paquetes pendientes en 10240, veremos la siguiente salida al detener la ejecución de Suricata:

```
^C4/12/2017 -- 12:19:23 - <Notice> - Signal Received.  Stopping engine.
4/12/2017 -- 12:19:26 - <Notice> - Stats for 'ens33':  pkts: 867380, drop: 797912 (91.99%), invalid chksum: 0
```

Esto quiere decir que la acción *reject* permitió que Suricata bloquee todos los paquetes que considera que son parte del ataque.

Analizamos el archivo *fast.log* y encontramos el registro de la regla.

```
12/04/2017-12:19:00.881550  [wDrop] [**]
[1:1000010:0] Posible ataque TCP DoS! [**]
[Classification: Posible DoS]
[Priority: 1] {TCP} 192.168.0.32:14936 -> 192.168.0.12:80
```

Sistema de prevención de intrusos (IPS)

En esta prueba seguiremos manteniendo la cantidad máxima de paquetes en 10240.

Primero añadimos la regla a *iptables* para que se analice el trafico TCP entrante hacia mi puerto 80 (también podría ser cualquiera si no se especifica ninguno).

```
ale@ubuntu:~$ sudo iptables -I INPUT -p tcp --dport 80 -j NFQUEUE
```

Añadimos una regla para que realice *drop* a los paquetes TCP que intenten realizar un DoS.

```
drop tcp any any -> $HOME_NET $HTTP_PORTS
(flags: S; msg:"Posible ataque TCP DoS!";
flow: to_server, stateless;
threshold: type both, track by_dst, count 70, seconds 10;
classtype:regla-DoS;
sid:1000015;)
```

Ejecutamos Suricata en modo IPS y observamos la regla que se añadió a *iptables*.

```
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 NFQUEUE    tcp  --  *      *       0.0.0.0/0            0.0.0.0/0           tcp dpt:80 NFQUEUE num 0

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
```

Desde *Kali Linux* realizamos el ataque DoS con el comando *hping3* como se vio con anterioridad. Al frenarlo podemos visualizar los paquetes que envió (739662).

```
HPING 192.168.0.12 (eth0 192.168.0.12): S set, 40 headers + 0 data bytes
hping in flood mode, no replies will be shown
^C
--- 192.168.0.12 hping statistic ---
739662 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
```

Luego, al visualizar la tabla *iptables* podemos notar que ingresaron 740K paquetes que se corresponden con el protocolo TCP:

```
Chain INPUT (policy ACCEPT 5 packets, 1232 bytes)
 pkts bytes target     prot opt in     out     source               destination
 740K   30M NFQUEUE    tcp  --  *      *       0.0.0.0/0            0.0.0.0/0           tcp dpt:80 NFQUEUE num 0

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 143 packets, 7160 bytes)
 pkts bytes target     prot opt in     out     source               destination
```

Al detener Suricata podemos verificar que la gran mayoría de los paquetes analizados son soltados (Dropped) debido a la regla agregada con anterioridad.

```
^C4/12/2017 -- 12:31:04 - <Notice> - Signal Received.  Stopping engine.
4/12/2017 -- 12:31:07 - <Notice> - (RX-Q0) Treated: Pkts 739521, Bytes 29580840, Errors 0
4/12/2017 -- 12:31:07 - <Notice> - (RX-Q0) Verdict: Accepted 138, Dropped 739383, Replaced 0
```

En el archivo *fast.log* podemos observar el registro de nuestra regla.

```
12/04/2017-12:30:52.010092  [Drop] [**]
[1:1000015:0] Posible ataque TCP DoS!
[**] [Classification: Posible DoS]
[Priority: 1] {TCP} 192.168.0.32:53129 -> 192.168.0.12:80
```

3.9.3: Escaneo de puertos

Se utilizará una herramienta llamada *nmap* cuya función consiste en evaluar la seguridad de sistemas informáticos para así poder detectar sus vulnerabilidades del sistema operativo, puertos abiertos, servicios que se ejecutan, etc. Además, tiene la capacidad de adaptarse a las condiciones de red incluyendo latencia y demás.

Con *nmap* se hará un escaneo de puertos, sistema operativo, versión y a Suricata para que este último lance una alerta advirtiendo las acciones que está realizando dicha herramienta.

Como primer paso, se utilizará una regla del archivo *emerging-scan.rules* perteneciente al conjunto de reglas de Emerging Threats que esta desactivada de forma predeterminada. Para simplificar su búsqueda se la añadió a nuestro archivo de reglas *local.rules*. Cabe aclarar que las reglas que

alertan sobre las diferentes acciones de *nmap* son una gran cantidad y a modo de simplicidad solo se utilizará la que se puede visualizar a continuación:

```
alert tcp $HOME_NET any -> $HOME_NET any
(msg:"ET SCAN NMAP -sA (1)";
fragbits:!D;
dsize:0;
flags:A,12;
window:1024;
threshold: type both, track by_dst, count 1, seconds 60;
reference:url,doc.emergingthreats.net/2000538;
classtype:attempted-recon;
sid:2000538;
rev:8;)
```

En la cual se analizarán todos los paquetes TCP que provienen desde cualquier IP y puerto hacia la red local en cualquier puerto. Los demás parámetros son explicados a continuación:

Fragbits:!D: es utilizado para modificar el mecanismo de fragmentación, en este caso con *D* no se realiza fragmentación y con *!* se hará match si los bits especificados no fueron utilizados.

Dsize:0: ocurrirá un match cuando la carga útil (*payload*) del paquete no tenga datos.

Flags: A, 12: A es un flag TCP que proviene de ACK y es utilizado para indicar que el campo de asentimiento es válido. Todos los paquetes enviados después del paquete SYN inicial deben tener activo este flag. Muchas herramientas que realizan escaneo de puertos envían paquetes que poseen este valor de *ack* a un determinado puerto para realizar tareas de reconocimiento. El valor 12 indica un SYN-ACK, esto se da cuando el objetivo le responde a un intento de sincronización SYN al emisor. A modo de resumen, lo que hace este parámetro es detectar paquetes que quieren sacar información al objetivo o simulan que el objetivo le envió un paquete con SYN.

Window:1024: tamaño de ventana TCP que indica el volumen de bytes que pueden ser recibidos sin hacer el *ack*.

Threshold: type both, track by_dst, count 1, seconds 60: dicho parámetro se utiliza para controlar la frecuencia de alertas de la regla. El tipo *both* combina el tipo *threshold* (establece un umbral a partir del cual generará una alerta), con el tipo *limit* (establece un límite al número de alertas generadas). El parámetro *track by_dst* especifica que se debe tener en cuenta hacia dónde va dirigida la conexión. En este caso, se generará una alerta si hay 1 o más paquetes TCP que cumplan las condiciones en 60 segundos.

La regla anterior pertenece a la clase *attempted-recon* de alta prioridad definida en el archivo *classification.config* de la siguiente manera:

```
config classification: attempted-recon,
Attempted Information Leak,2
```

Sistema de detección de intrusos (IDS)

Para comenzar la prueba ejecutamos Suricata en modo IDS como vimos en los tests anteriores.

Ahora desde el Kali Linux ejecutamos *nmap* con el parámetro *-sA* para realizar un análisis TCP ACK y colocamos la IP de Suricata como objetivo:

```
root@kali:~# nmap -sA 192.168.0.18
```

Y al finalizar se muestra un reporte del escaneo incluyendo la cantidad de puertos cerrados e información sobre los abiertos:

```
Starting Nmap 7.50 ( https://nmap.org ) at 2017-07-23 11:35 -03
Nmap scan report for 192.168.0.18
Host is up (0.00057s latency).
All 1000 scanned ports on 192.168.0.18 are unfiltered
MAC Address: 00:0C:29:75:3E:24 (VMware)

Nmap done: 1 IP address (1 host up) scanned in 3.50 seconds
```

Luego, detenemos la ejecución de Suricata y al observar el archivo *fast.log* podremos notar como se alertó del intento de fuga de información por parte de la dirección IP ilustrada:

```
07/23/2017-11:35:41.419778  [**] [1:2000538:8] ET SCAN NMAP -sA (1)
[**] [Classification: Attempted Information Leak]
[Priority: 2] {TCP} 192.168.0.105:58115 -> 192.168.0.18:21
```

3.9.4: Ransomware

En los últimos años los sistemas informáticos han sido expuestos a una nueva forma de ataque a través de un tipo de malware llamado *ransomware*. Este tiene la capacidad de infectar un equipo, bloquearlo y/o cifrar sus datos, para luego pedir al usuario del equipo una suma importante de dinero como "rescate" para su "recuperación". Es importante mencionar que el pago del rescate no implica que los datos sean recuperados, es por ello que no se recomienda realizar el mismo.

Suricata es una herramienta apropiada para proteger a un sistema contra malware y muy efectiva contra ransomware.

En esta sección se realizará un test con dos tipos de ransomware que ocasionaron importantes daños ya sea tanto a empresas como a personas:

- *CryptoWall*: descubierto en septiembre de 2014. Su expansión a miles de computadoras se realizó a través de correos spam, anuncios con contenido malicioso, sitios de internet comprometidos, etc. Una vez ejecutado, crea una serie de entradas de registro para almacenar la ruta de los archivos cifrados y para poder iniciarse junto con el sistema operativo. A través de una clave de cifrado RSA 2048 codifica los archivos con extensión *.doc, .docx, .xls, .ppt, .psd, .pdf, .jpg,* entre otros; y crea unos archivos de texto con las instrucciones sobre cómo realizar el pago para obtener la clave de descifrado.
- *PetrWrap*: variante del malware *Petya*, su expansión comenzó en junio de 2017. Utiliza un *exploit* llamado *EternalBlue* que aprovecha una vulnerabilidad del protocolo de red SMB[9] (*Server Message Block*) de Windows para sobrescribir el registro de arranque principal (o **MBR**[10]) y disparar un reinicio del sistema para su posterior ejecución. Luego, cifra la tabla

[9] *Server Message Block*, es un protocolo de red que permite compartir archivos, impresoras entre nodos de una red de computadoras que usan el sistema operativo Windows.
[10] *Master Boot Record*, es el primer sector de un dispositivo de almacenamiento de datos, como un disco duro.

maestra de archivos del sistema de archivos NTFS mientras muestra un falso mensaje del escáner de discos *chkdsk*[11].

Se cree que dicho exploit fue creado por la Agencia de Seguridad Nacional (NSA) de los Estados Unidos y filtrado por un grupo de *hackers*.

Su ataque más importante registrado hasta la fecha fue contra el Gobierno de Ucrania ocasionando daños irreversibles en sistemas de archivos de aeropuertos, bancos y desactivando sistemas de monitoreo como el de radiación en la planta nuclear de Chernobyl.

Es importante mencionar que gran parte de la efectividad de este tipo de malware se centra en tres aspectos fundamentales:

- La utilización de algoritmos de cifrado basados en problemas matemáticos, los cuales tienen una gran eficacia en su aplicación y un descifrado completo que es computacionalmente intratable. Es por esto que la defensa antes este tipo de malware se centra en su prevención y en la búsqueda de vulnerabilidades encontradas en el ransomware, como puede ser el lugar donde se guardan las claves de descifrado.
- La imposibilidad de rastrear a sus autores ya que el pago siempre se solicita a través de una red anónima llamada *tor* que mantiene la integridad y el secreto de la información que viaja por ella.
- El pago es solicitado en un tipo de moneda digital llamada *bitcoin* que no está respaldado por ningún gobierno ni es posible su seguimiento. El costo del bitcoin ha aumentado de manera significativa en los últimos años, a modo de ejemplo 1 bitcoin equivale a US$6567 en octubre de 2017.

Debido a la peligrosidad de este tipo de malware, solo será testeada su prevención ya que una alerta dejaría actuar al mismo y una vez que se realiza el cifrado es demasiado tarde.

Para ambos, se realizará la detección de un archivo comprimido en formato *zip*. Además, en el caso de *CryptoWall*, de un archivo binario que resulta de la extracción y, en el de *PetrWrap*, de un archivo ejecutable infectado.

Como primer paso creamos la clase con una alta prioridad en el archivo *classification.config* que contendrá nuestras cuatro reglas para la detección de cada malware.

```
config classification: regla-ransomware,Posible ataque de Ransomware,1
```

Ahora se prosigue con la creación de las reglas. Para estas, lo ideal hubiera sido utilizar el parámetro *pcre* con expresiones regulares que permitan capturar varias características de los archivos maliciosos con mayor abstracción y así, ser menos vulnerables a cambios mínimos en el malware que se originan para intentar sobreponerse a los controles. Sin embargo, muchas veces esto no es posible debido a que se necesita el código fuente del archivo y rara vez se encuentra disponible. Por lo tanto, no queda otra opción más que la de usar el parámetro *content* y capturar conjuntos de dígitos hexadecimales elegidos de forma aleatoria de cada archivo. Muchas veces

[11] Comando utilizado para comprobar la integridad tanto de unidades de disco duro como unidades de disco flexible, y para reparar errores lógicos en el sistema de archivos de Windows.

ocurre que en este tipo de sistemas de prevención y detección de intrusos existe un límite temporal muy acotado por lo que el objetivo es buscar una solución de forma inmediata por más de que sea muy específica. Una forma de disminuir las desventajas que la detección por contenido posee es la de combinar varias estrategias como las siguientes:

- Realizar el monitoreo de malware de forma continua para crear reglas efectivas y así compartirlas a la comunidad. Tengamos en cuenta que la actualización de reglas es diaria.
- Utilizar varios *content* por regla para capturar diferentes características de un malware y hacer una detección más genérica.
- Crear varias reglas para un mismo malware con el objetivo de aumentar su probabilidad de detección.
- Estudiar aspectos relacionados con el comportamiento del malware o de red para luego ser plasmados en reglas.

A modo de simplificar la creación de reglas se especificará un solo *content* por regla con 32 pares de dígitos hexadecimales consecutivos extraídos de una ubicación aleatoria de cada archivo. Para esto se utiliza una herramienta llamada *hexdump*. Por ejemplo, tomamos los dígitos del archivo ejecutable de PetrWrap de la siguiente manera:

```
ale@ubuntu:~$ hexdump -C svchost.exe
```

Parte de la salida será:

```
000488a0  01 00 3e 1d 01 20 01 00  20 2b 11 20 02 00 28 2b  |..>.. .. +. ..(+|
000488b0  11 20 03 00 34 2b 01 20  01 00 d4 2a 11 20 02 00  |. ..4+. ...*. ..|
000488c0  47 2b 11 20 03 00 55 2b  11 20 04 00 3e 1d 00 20  |G+. ..U+. ..>.. |
000488d0  00 00 00 00 01 20 01 00  6a 2b 00 20 00 00 00 00  |..... ..j+. ....|
000488e0  01 20 01 00 8c 2b 00 20  00 00 00 00 01 20 01 00  |. ...+. ..... ..|
000488f0  98 2b 01 20 01 00 98 2b  01 20 02 00 a6 2b 11 20  |.+. ...+. ...+. |
00048900  03 00 3e 1d 01 20 01 00  98 2b 01 20 01 00 3e 1d  |..>.. ...+. ..>.|
00048910  01 20 02 00 ee 2b 11 30  03 00 f6 2b 01 20 01 00  |. ...+.0...+. ..|
```

Las reglas estarán dirigidas a realizar *drop* de los datos transmitidos bajo el protocolo TCP originados en el puerto 80 de un IP cualquiera hacia el conjunto de IPs pertenecientes a la red local a cualquier puerto.

Regla para detectar CryptoWall en archivo comprimido *zip*:

```
drop tcp any 80 -> $HOME_NET any
(msg: "Ransomware Cryptowall comprimido en zip detectado";
content:"|24 8d 02 06 f8 2b 87 24 be 33 87 15 9e 70 2b cf
a6 20 81 92 c9 27 bf 14 f8 9e d4 3e bf b7 7b fa|";
classtype:regla-ransomware;
sid:1000008;)
```

Regla para detectar CryptoWall en archivo binario:

```
drop tcp any 80 -> $HOME_NET any
(msg: "Ransomware Cryptowall en archivo bin detectado";
content:"|57 2b 39 42 73 42 2f 54 74 2f 6c 47 4a 51 43
61 63 4c 4f 6f 6c 64 6f 4d 49 33 2b 44 78 59 66 55|";
classtype:regla-ransomware;
sid:1000007;)
```

Regla para detectar PetrWrap en archivo comprimido *zip*:

```
drop tcp any 80 -> $HOME_NET any
(msg: "Ransomware Petrwrap comprimido en zip detectado";
content:"|f1 12 0b e5 b7 f5 e0 ba f5 a8 e7 37 0d d1 ed
cb 03 48 ee d6 7d 0d 79 2e ab af ad 91 3a ee 25 be|";
classtype:regla-ransomware;
sid:1000006;)
```

Regla para detectar ejecutable de PetrWrap:

```
drop tcp any 80 -> $HOME_NET any
(msg: "Ransomware Petrwrap en svchost detectado";
content:"|81 cc 07 00 02 01 12 82 14 0a 07 00 02
01 12 82 2c 0e 07 00 02 01 12 82 c0 0e 06 00 01 12 82 20|";
classtype:regla-ransomware;
sid:1000005;)
```

Para la realización del test se levantará un servidor *apache2* [12]en el Kali Linux en el que se venían realizando las pruebas anteriores. Viene incluido con la distribución Kali Linux, por lo que no se requiere instalación. Posee un archivo *index.html* ubicado en */var/www/html* que contiene la página predeterminada en la dirección 127.0.0.1 (*localhost*) y que sirve para verificar que el servidor fue levantado correctamente.

Se utilizó esta herramienta para crear un simple servidor de archivos que alojará los cuatro malware en la carpeta *Ransomware* ubicada en la ruta */var/www/html/Ransomware*.

```
root@kali:/var/www/html/Ransomwares# ls
cryptowall.bin  Ransomware.Cryptowall.zip  Ransomware.Petrwrap.zip  svchost.exe
```

Acto seguido, se modificó el *index* establecido para añadir los links de descarga a los archivos de la siguiente manera:

```
<!doctype html>

<html lang="en">
<head>
  <meta charset="utf-8">

  <title>Ransomware Test</title>
  <meta name="description" content="The HTML5 Herald">
  <meta name="Alexis Fredes" content="Ransomware Test">

  <link rel="stylesheet" href="css/styles.css?v=1.0">

</head>

<body>

  <p>Descarga el archivo:</p>
<a href="/Ransomware/cryptowall.bin" download>
        <p>CryptoWall bin</p>
        </a>
<a href="/Ransomware/Ransomware.Cryptowall.zip" download>
        <p>CryptoWall zip</p>
        </a>
<a href="/Ransomware/Ransomware.Petrwrap.zip" download>
        <p>Petrwrap zip</p>
        </a>
<a href="/Ransomware/svchost.exe" download>
        <p>Petrwrap svchost.exe</p>
        </a>
</body>
</html>
```

[12] Apache es un servidor HTTP de código abierto para plataformas Unix, Microsoft y Macintosh. Es uno de los servidores más ampliamente utilizados en el mundo debido a su modularidad, extensibilidad, popularidad, entre otros.

Sistema de prevención de intrusos (IPS)

Añadimos la regla a *iptables* para que el IPS analice el trafico TCP entrante proveniente del puerto 80.

```
ale@ubuntu:~$ sudo iptables -I INPUT -p tcp --sport 80 -j NFQUEUE
```

Quedando de la siguiente manera:

```
ale@ubuntu:~$ sudo iptables -vnL
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 NFQUEUE    tcp  --  *      *       0.0.0.0/0            0.0.0.0/0           tcp spt:80 NFQUEUE num 0

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
```

Una vez finalizada la etapa de configuración se comienza con el test. Primero levantamos el servidor *apache2* en el Kali Linux:

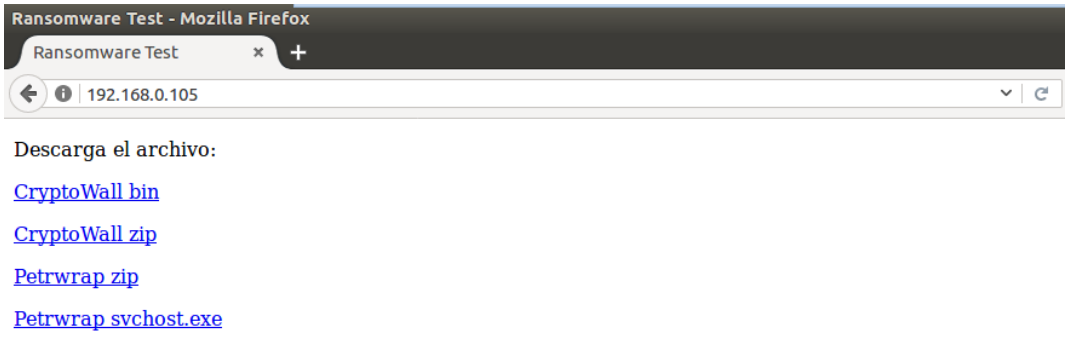
```
root@kali:~# /etc/init.d/apache2 start
[ ok ] Starting apache2 (via systemctl): apache2.service.
root@kali:~#
```

Luego, averiguamos la IP de Kali Linux con el comando *ifconfig*, en este caso es *192.168.0.105*.

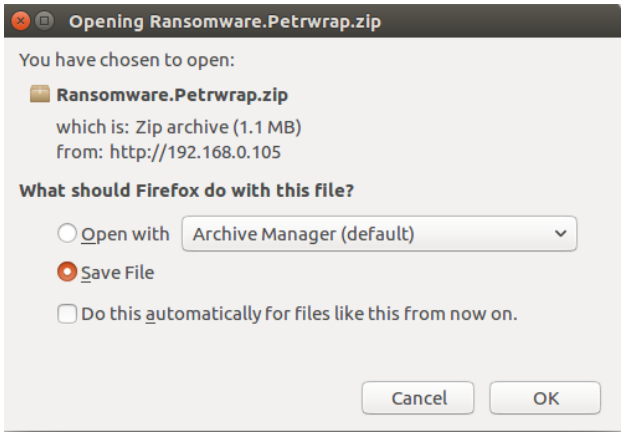
Ahora vamos a Suricata y ejecutamos el IPS:

```
ale@ubuntu:~$ sudo LD_PRELOAD="/usr/lib/libtcmalloc_minimal.so.4" suricata -c /etc/suricata/suricata.yaml -q 0
[sudo] password for ale:
21/7/2017 -- 18:04:06 - <Notice> - This is Suricata version 3.2.1 RELEASE
21/7/2017 -- 18:04:29 - <Notice> - all 6 packet processing threads, 4 management threads initialized, engine started.
```

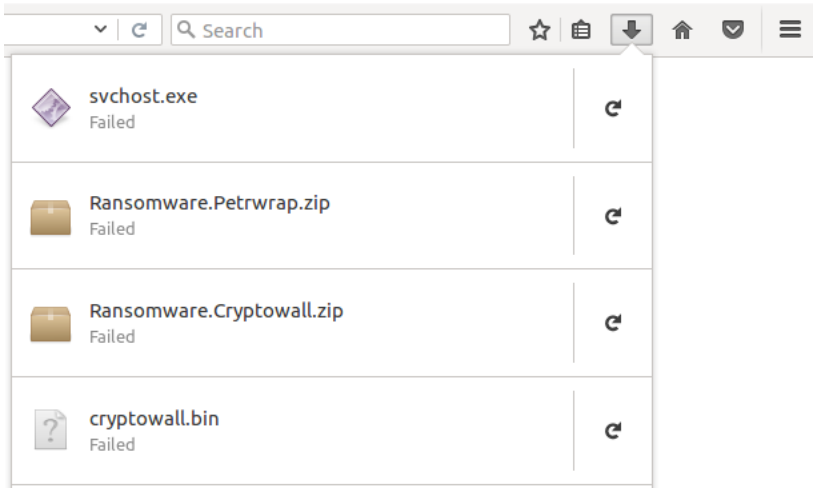
Luego abrimos el navegador (en este caso *Firefox*), nos dirigimos a la IP de nuestro servidor LAN y se mostrará el *index.html* con los ransomware para descargar.



Procedemos a la descarga de los cuatro archivos haciendo clic en cada uno de ellos:



Y vamos a poder apreciar que las descargas no se van a realizar ya que Suricata analizo el contenido de cada una y aplico la acción *drop* luego que cada archivo hiciera match con cada una de las reglas:



Por último, al detener la ejecución de Suricata podemos visualizar que entre los paquetes analizados hubo 193 que no permitieron:

```
^C21/7/2017 -- 18:13:54 - <Notice> - Signal Received.  Stopping engine.
21/7/2017 -- 18:13:55 - <Notice> - (RX-Q0) Treated: Pkts 1477, Bytes 1718935, Errors 0
21/7/2017 -- 18:13:55 - <Notice> - (RX-Q0) Verdict: Accepted 1284, Dropped 193, Replaced 0
```

Al analizar el archivo *fast.log* también podemos apreciar la acción tomada para cada archivo:

```
07/21/2017-18:10:22.766320  [Drop] [**] [1:1000007:0] Ransomware Cryptowall en archivo bin detectado
[**] [Classification: Posible ataque de Ransomware]
[Priority: 1] {TCP} 192.168.0.105:80 -> 192.168.0.18:33100

07/21/2017-18:11:20.109838  [Drop] [**] [1:1000008:0] Ransomware Cryptowall comprimido en zip detectado
[**] [Classification: Posible ataque de Ransomware]
[Priority: 1] {TCP} 192.168.0.105:80 -> 192.168.0.18:33102

07/21/2017-18:12:17.451637  [Drop] [**] [1:1000006:0] Ransomware Petrwrap comprimido en zip detectado
[**] [Classification: Posible ataque de Ransomware]
[Priority: 1] {TCP} 192.168.0.105:80 -> 192.168.0.18:33104

07/21/2017-18:12:39.979248  [Drop] [**] [1:1000005:0] Ransomware Petrwrap en svchost detectado
[**] [Classification: Posible ataque de Ransomware]
[Priority: 1] {TCP} 192.168.0.105:80 -> 192.168.0.18:33106
```

Mientras que para el *eve.json* a modo de referencia una salida será:

```
{
 "timestamp":"2017-07-21T18:11:46.732894-0300",
 "flow_id":723409627319985,
 "event_type":"alert",
 "src_ip":"192.168.0.105",
 "src_port":80,
 "dest_ip":"192.168.0.18",
 "dest_port":33106,
 "proto":"TCP",
 "alert":{
  "action":"blocked",
  "gid":1,
  "signature_id":1000005,
  "rev":0,
  "signature":"Ransomware Petrwrap en svchost detectado",
  "category":"Posible ataque de Ransomware",
  "severity":1
 }
}
```

Capítulo 4: Machine Learning

4.1: Definición

Machine Learning o Aprendizaje Automático es una rama de la Inteligencia Artificial[13] que permite que las computadoras tengan la capacidad de aprender sin que ello este explícitamente programado. Para esto, se explora el estudio y la construcción de algoritmos que pueden aprender de los datos y hacer predicciones en base a los mismos.

El aprendizaje es una parte esencial de la inteligencia ya que brinda la posibilidad de mejorar el comportamiento utilizando la experiencia propia:

- El rango de comportamiento se expande, esto quiere decir que se adquiere la capacidad de hacer más.
- Se mejora la precisión con que se ejecutan tareas, ya que por medio del aprendizaje se puede mejorar el desempeño.
- Se acelera la ejecución de las tareas.

En la definición formal se considera que una maquina "aprende" sobre una tarea T, una métrica de rendimiento R y un tipo de experiencia E; si nos aseguramos que mejora su rendimiento R en la tarea T siguiendo la experiencia E.

Las técnicas de *machine learning* se basan en establecer un modelo explícito o implícito que permite que los patrones analizados sean categorizados. Una característica distintiva de estos esquemas es la necesidad de conjuntos de datos para entrenar el modelo de comportamiento. Dicho procedimiento requiere una gran demanda de recursos.

En muchos casos, la aplicabilidad de los principios de *machine learning* coinciden con la de muchas técnicas basadas en modelos estadísticos. Sin embargo, *machine learning* se centra en la construcción de un modelo que mejora su rendimiento a partir de resultados previos. Por lo tanto, su aplicación en los Sistemas de Detección de Intrusos les brinda la posibilidad de modificar la estrategia de ejecución a medida que se obtiene información nueva.

Suelen clasificarse en tres grandes categorías, dependiendo de la "retroalimentación" (feedback) que recibe el sistema:

- Aprendizaje supervisado: un "maestro" le brinda al programa las entradas y las salidas que se desean obtener a partir de dichas entradas. El objetivo es el aprendizaje de una regla general que realice el mapeo de las entradas a las salidas.
- Aprendizaje sin supervisión: el programa aprende patrones sobre la entrada, aunque no se suministre ninguna información explicita o etiquetado de la salida esperada en cada caso.

[13] Según John McCarthy, es la Ciencia e Ingeniería de construir artefactos inteligentes, en especial programas inteligentes de computadora. También se relaciona con el uso de computadoras para entender la inteligencia humana.

- Aprendizaje por refuerzo: el programa interactúa con un entorno dinámico en el que debe realizar una tarea. Una vez realizada la tarea el programa recibe datos como retroalimentación en términos de recompensas y castigos.

4.2: Aplicabilidad en los Sistemas de Detección de Intrusos

El primer interrogante es si existe realmente la necesidad de aplicar inteligencia artificial a un sistema de detección de intrusos.

Por un lado, se puede afirmar que el proyecto *Emerging Threats*, utilizado para obtener las reglas que usamos en Suricata, tiene como una de sus fuentes de malware resultados provenientes del uso de herramientas de *Machine Learning*. Esto le permite estar muy bien preparado para detectar nuevas amenazas y, consecuentemente crear las reglas que permitan su detección.

Reportes llevados a cabo a mediados de 2017 por la Universidad de San Diego a partir del análisis de una red de gran tamaño con pocos controles perimetrales y con una gran cantidad de dispositivos móviles conectados pertenecientes a personas que visitan el lugar (o **BYOD** de sus siglas en inglés *Bring Your Own Device*) revelan que solo se detectó un 30% de la totalidad de señales únicas pertenecientes a *Emerging Threats PRO*. Esto no es suficiente para afirmar que el comportamiento será similar para cualquier otro sistema, pero nos da un indicio de que en dicho caso existe un amplio margen entre las amenazas que se buscan y las que fueron encontradas. Por lo tanto, se podría llegar a afirmar que el uso de técnicas de inteligencia artificial no es necesario.

Por otro lado, incluir técnicas de inteligencia artificial podría dar lugar a múltiples beneficios como el de la creación de reglas personalizadas en base a nuestra detección y así evitar un posible ataque dirigido solo a nuestro sistema. También se podrían obtener reglas de manera anticipada a los repositorios conocidos y se podría mejorar la eficiencia en la detección a medida que el sistema realiza el aprendizaje.

Personalmente, pienso que el mejor enfoque sería uno *hibrido*, donde por un lado se obtienen reglas de los repositorios y por otro se crean reglas adaptadas a nuestro sistema en particular.

Numerosos esquemas basados en *Machine Learning* son aplicados a Sistemas de Detección de Intrusos basados en Heurísticas, los más importantes son:

- Redes Bayesianas: es un modelo codifica relaciones basadas en probabilidad entre variables de interés. Generalmente se aplica a la detección de intrusos en combinación con esquemas estadísticos.
  Como ventajas de este esquema se pueden resaltar la capacidad de codificación de interdependencias entre variables, la predicción de eventos y la capacidad de incorporar datos y conocimiento previo.
  Como desventaja tenemos que, si bien sus resultados son equiparables a aquellos derivados de los sistemas basados en umbral, se requiere un procesamiento computacional mucho mayor. Además, los resultados que se obtienen con redes bayesianas son altamente dependientes de las suposiciones hechas sobre el

comportamiento del sistema objetivo, por lo que una pequeña desviación en dichas hipótesis puede dar lugar a errores de detección.

- Modelos de Markov: en esta categoría se definen dos enfoques: Cadenas de Markov y Modelos Ocultos de Markov. Una cadena de Markov es un conjunto de estados que están interconectados a través de ciertas probabilidades de transición que determinan la topología y las capacidades del modelo. Durante la primera fase de entrenamiento, las probabilidades asociadas a las transiciones son estimadas a partir del comportamiento normal del sistema objetivo. Luego, la detección de anomalías es llevada a cabo en base a la comparación entre cantidad de anomalías obtenidas a partir de las secuencias observadas y un umbral fijo.
  En el caso del Modelo Oculto de Markov, se asume que el sistema de interés es un proceso de Markov que posee los estados y transiciones ocultas.
- Redes Neuronales: con el propósito de simular el funcionamiento del cerebro humano (en especial el de las neuronas y sinapsis) se adoptan redes neuronales debido a su gran flexibilidad y adaptación a los cambios en el entorno. Este enfoque ha sido utilizado para crear perfiles de usuario, para predecir el siguiente comando en base a los comandos vistos, y para identificar un comportamiento intrusivo en los patrones generados a partir de tráfico.
- Técnicas de Lógica Difusa: la lógica difusa se deriva de la teoría de conjuntos difusos bajo la cual el razonamiento es aproximado en vez de deducirse de forma precisa como sucede en la lógica de predicados clásica. Aunque la lógica difusa posee una gran efectividad contra escaneos de puertos e intentos de reconocimientos, su principal desventaja es el gran consumo de recursos que involucran sus acciones.
- Algoritmos Genéticos: utilizan técnicas inspiradas en la evolución como pueden ser herencia, mutación, selección y recombinación. A partir de ellos, es capaz de derivar reglas de clasificación y seleccionar las características apropiadas para el proceso de detección.
  Su mayor ventaja es la utilización de un método flexible, robusto y global de búsqueda que converge a una solución a partir de múltiples direcciones, teniendo en cuenta que no se asume conocimiento previo sobre el comportamiento del sistema.
  Como sucede con la mayoría de los algoritmos, su principal desventaja es el gran consumo de recursos.
- Árbol de decisión: es un modelo de predicción que será explicado de forma detallada en la siguiente sección.

4.3: Árboles de Decisión

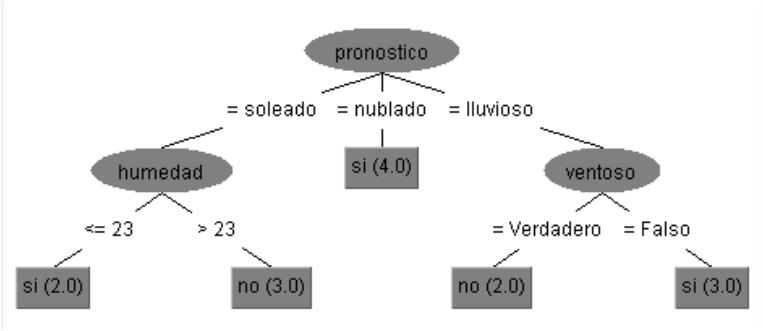
El aprendizaje basado en la construcción de árboles de decisión se clasifica dentro de la categoría de aprendizaje supervisado. En el aprendizaje supervisado, el programa observa un conjunto de entrenamiento dado por un “maestro”, que corresponden a pares (input-output), esto es, entradas de ejemplo y sus salidas deseadas.

La meta es aprender una regla general que mapea entradas a salidas. Esa regla se aplicará en el futuro a otras entradas y deberá generar una salida.

Un árbol de decisión es una herramienta para aprender de los ejemplos observados y poder dar una respuesta ante ejemplos similares en el futuro.

Un árbol de decisión es un árbol donde:

- Los nodos internos se etiquetan con atributos que caracterizan a los datos o experiencias.
- Los arcos salientes de un nodo interno etiquetado con el atributo A se etiquetan con cada uno de los posibles valores del atributo A.
- Las hojas del árbol se etiquetan con clasificaciones (ej.: si / no).



Se asume que los atributos tienen un rango de valores finito y discreto.

Expresividad de los árboles de decisión

La lógica que representa a un árbol de decisión booleano es equivalente a una aserción en la que el atributo meta es verdadero si y solo si todos los atributos de entrada satisfacen uno de los caminos comenzando en la raíz y finalizando en un nodo hoja con resultado positivo. Expresando esto en lógica proposicional tenemos:

$$Meta \leftrightarrow (Camino_1 \vee Camino_2 \vee \dots),$$

Donde cada camino es una conjunción de los test atributo-valor requeridos para seguir dicho camino.

Los árboles de decisión pueden expresar exactamente lo mismo que los lenguajes de tipo proposicional, es decir, cualquier función booleana puede ser escrita como un árbol de decisión. Sin embargo, existen funciones booleanas, como la de paridad, que requieren un árbol de decisión de crecimiento exponencial por lo que utilizar árboles de decisión en dicho caso no sería la mejor opción.

Inducir árboles de decisión a partir de ejemplos

Un ejemplo para un árbol de decisión booleano consiste en un vector de atributos de entrada, *X*, y un único valor de salida booleano *y*. El conjunto de ejemplos completo se denomina *conjunto de entrenamiento*. Para que un árbol sea consistente con el conjunto de entrenamiento solo se necesitaría un árbol de decisión que posea un camino hasta una hoja para cada ejemplo, el problema con ello es que como el árbol memoriza las observaciones no podrá decir mucho sobre cualquier otro caso.

La idea principal detrás del árbol de decisión es que extraiga uno o más patrones a partir de los ejemplos para poderlos aplicar en situaciones desconocidas. Para esto, se debería encontrar el árbol más pequeño que sea consistente con los ejemplos. El problema radica en que para cualquier definición razonable encontrar el árbol más pequeño es un problema intratable por lo que solo se

podrá encontrar un árbol pequeño gracias al uso de determinadas heurísticas. El algoritmo de aprendizaje del árbol de decisión adopta la estrategia divide y vencerás, esto quiere decir que siempre se debe realizar primero el test al atributo más importante. Se considera como más importante aquel que discrimina más claramente a los ejemplos. Dicho test dividirá el problema en pequeños sub-problemas que a su vez serán resueltos en forma recursiva considerando los siguientes cuatro casos:

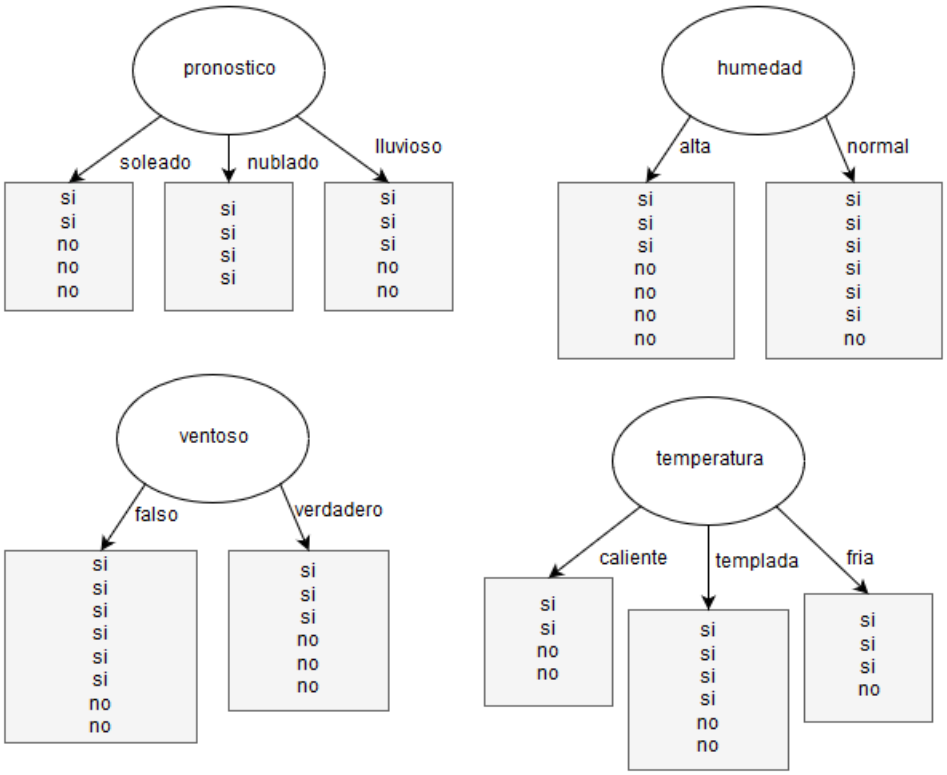
1. Si quedan ejemplos positivos y negativos hay que elegir el mejor atributo para separarlos.
2. Si todos los ejemplos restantes son positivos o negativos, se ha finalizado por lo que se puede responder "Si" o "No".
3. Si no quedan ejemplos, significa que ningún nodo ha sido observado y se devuelve el valor del nodo padre.
4. Si no quedan atributos, pero si ejemplos positivos y negativos se puede concluir que hay ruido en los datos. Esto ocurre cuando los atributos no brindan suficiente información para describir la situación.

Elección de los atributos de test

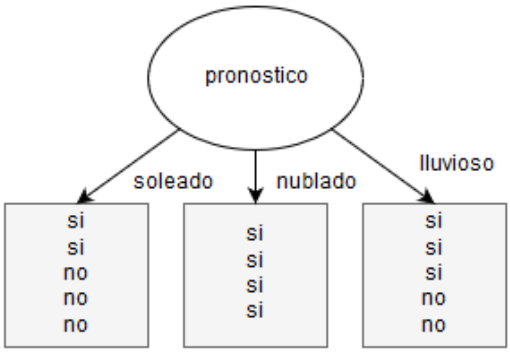
Es necesario discriminar los atributos en función de la cantidad de información que proporcionan para poder elegir al más indicado. La teoría de la información mide el contenido de la información en *bits* mediante el uso de la siguiente formula llamada entropía:

$$entropy(p_1, p_2, \dots, p_n) = -p_1 \log p_1 - p_2 \log p_2 \dots - p_n \log p_n$$

Con dicha ecuación se calcula la entropía de cada rama que nace en el atributo que estamos evaluando. Luego, se multiplica cada valor con una fracción que representa la cantidad de elementos de esa rama con respecto a todos los elementos y se suman los resultados para obtener la información esperada de todos los atributos, o también llamado el resto. Por último, se calcula la ganancia que será la información esperada de manera conjunta, esto es tomando todos los atributos como si estuvieran en una sola rama, y se le quita el resto. A modo de ejemplo, si tenemos que seleccionar el óptimo de los siguientes árboles:



Tomamos el primer árbol ya que a simple vista es el que toma decisión más definida con respecto a los demás:



Calculamos la entropía para la rama *soleado*, *nublado* y *lluvioso*, respectivamente:

$$entropy(2,3) = -\frac{2}{5}\log\frac{2}{5} - \frac{3}{5}\log\frac{3}{5} = 0.97095$$

$$entropy(4,0) = -1\log 1 = 0$$

$$entropy(3,2) = -\frac{3}{5}\log\frac{3}{5} - \frac{2}{5}\log\frac{2}{5} = 0.97095$$

La información esperada para los atributos se calcula de la siguiente forma:

$$entropy(2,3;4,0;3,2) = \frac{5}{14} * 0.971 + \frac{4}{14} * 0 + \frac{5}{14} * 0.971 = 0.693$$

Y la ganancia será:

$$ganancia(pronostico) = entropy(9{,}5) - \ entropy(2{,}3; 4{,}0; 3{,}2) \ = 0.940 - 0.693 = 0.247$$

Después, se debe calcular $ganancia(temperatura)$, $ganancia(humedad)$, $ganancia(ventoso)$ y seleccionar el atributo de mayor valor.

Calidad de los algoritmos de aprendizaje

Un algoritmo de aprendizaje es efectivo si sus hipótesis hacen buenas predicciones al pronosticar clasificaciones de ejemplos que no han sido observados denominados *conjunto de test*. Para saber la calidad de las predicciones, se contrastan las mismas con la clasificación correcta. Se recomienda seguir la siguiente metodología:

1. Recolectar un gran conjunto de ejemplos.
2. Dividir el conjunto anterior en dos conjuntos disjuntos: el de entrenamiento y el de test.
3. Aplicar el algoritmo de aprendizaje al conjunto de entrenamiento para generar la hipótesis *h*.
4. Medir el porcentaje de ejemplos del conjunto de test que *h* clasifica correctamente.
5. Repetir los pasos de 1 a 4 para conjuntos de entrenamiento de diferentes tamaños y conjuntos de entrenamiento seleccionados de forma aleatoria para cada tamaño.

Como resultado de la metodología antes vista tendremos un conjunto de datos que pueden ser procesados para obtener la calidad media de la predicción como una función del tamaño del conjunto de entrenamiento. A medida que es conjunto de entrenamiento crece, la calidad de la predicción será mayor.

Es importante destacar que al algoritmo de aprendizaje no se le debe permitir "visualizar" el conjunto de datos de test hasta que se utilice para contrastar con la hipótesis aprendida.

4.4: Experimentación con Árboles de Decisión

A partir de lo estudiado en los árboles de decisión, se aplicará esta forma de aprendizaje inductivo para lograr que el sistema de detección de intrusos logre predecir sus acciones luego de analizar conjuntos de datos nunca antes vistos.

En primer lugar, debemos obtener los paquetes de red para que sean analizados mediante una herramienta llamada *Wireshark*[14].

Nos dirigimos a nuestra máquina virtual e iniciamos nuestro *Ubuntu* para instalar *Wireshark*.

Primero actualizamos los repositorios:

```
ale@ubuntu:~$ sudo apt-get update
```

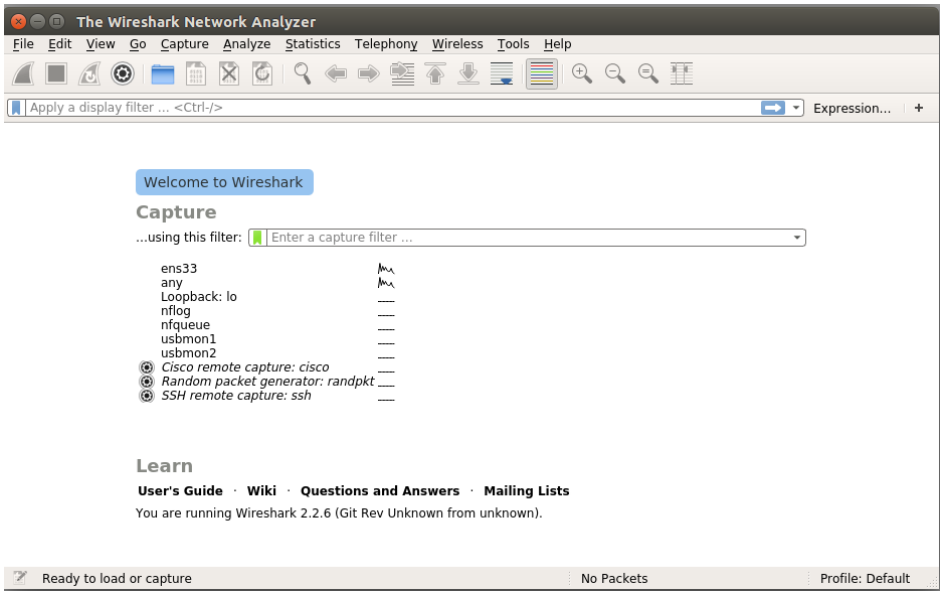
[14] La función principal de dicha herramienta *open source* es la de capturar y analizar paquetes de red.

Instalamos *Wireshark* siguiente comando:

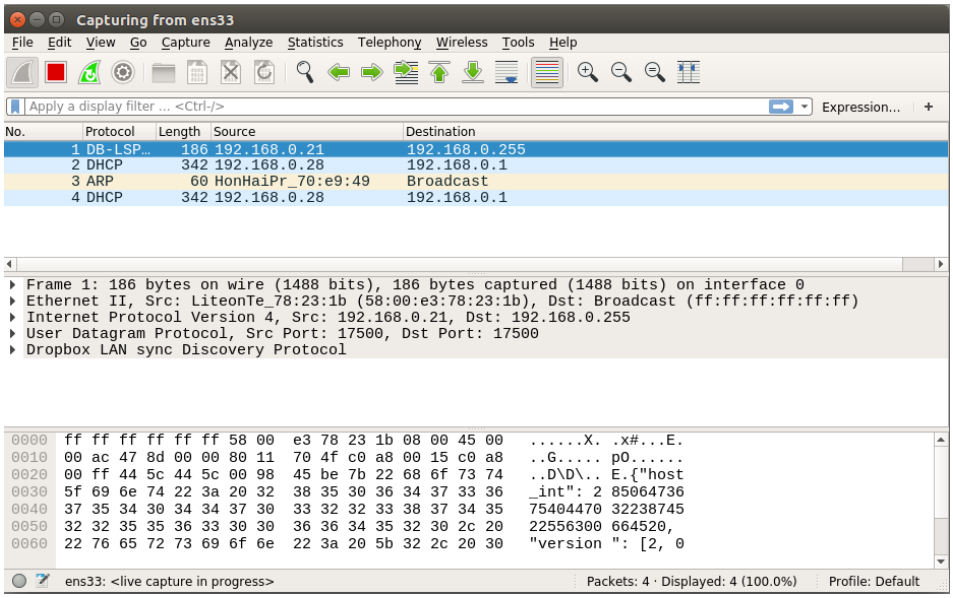
```
ale@ubuntu:~$ sudo apt-get install wireshark
```

Se explican dos opciones para obtener dichos paquetes:

1. Se puede realizar una captura de paquetes con la herramienta *Wireshark*. Lo ejecutamos con permisos de administrador desde la consola con "*sudo wireshark*" y veremos una interfaz similar a la siguiente:



Luego, seleccionamos nuestra interfaz de red, en este caso *ens33*, y comenzará la captura de paquetes.



Después de haber capturado unos 150.000 paquetes como mínimo detenemos la captura y la guardamos como "*captura-paquetes-pcap-X.pcap*".

2. Se pueden descargar los conjuntos de datos desde páginas web como la siguiente: https://mcfp.weebly.com . Esto sirve a modo de ejemplo ya que permite ahorrarnos el tiempo de la captura y además nos asegura que los paquetes estén infectados. Es importante mencionar que dichos paquetes serán usados más adelante para el aprendizaje del sistema de detección de intrusos por lo que su aplicación más efectiva estará en la red donde ocurrió la captura. En el ejemplo siguiente se usará esta opción ya que, al no tener como objetivo ser aplicado en un sistema real no tiene importancia el origen de los paquetes. De otro modo se debería utilizar la opción restante.

   El conjunto de datos elegido proviene de la página mencionada con anterioridad y es el llamado *CTU-13*. Contiene tráfico capturado por *botnets*[15] en la Universidad CTU de Republica Checa, en 2011. El objetivo de este conjunto de datos es el de obtener una gran cantidad de tráfico normal mezclado con tráfico de segundo plano. Consiste de trece escenarios de captura que registran el comportamiento de diferentes tipos de malware, protocolos utilizados y las acciones realizadas.
   En la siguiente tabla se pueden apreciar los distintos escenarios:

Table 2 – Characteristics of the botnet scenarios. (CF: ClickFraud, PS: Port Scan, FF: FastFlux, US: Compiled and controlled by us.)

Id	IRC	SPAM	CF	PS	DDoS	FF	P2P	US	HTTP	Note
1	√	√	√							
2	√	√	√							
3	√			√				√		
4	√				√			√		UDP and ICMP DDoS.
5		√		√					√	Scan web proxies.
6				√						Proprietary C&C. RDP.
7									√	Chinese hosts.
8				√						Proprietary C&C. Net-BIOS, STUN.
9	√	√	√	√						
10	√				√			√		UDP DDoS.
11	√				√			√		ICMP DDoS.
12							√			Synchronization.
13		√		√					√	Captcha. Web mail.

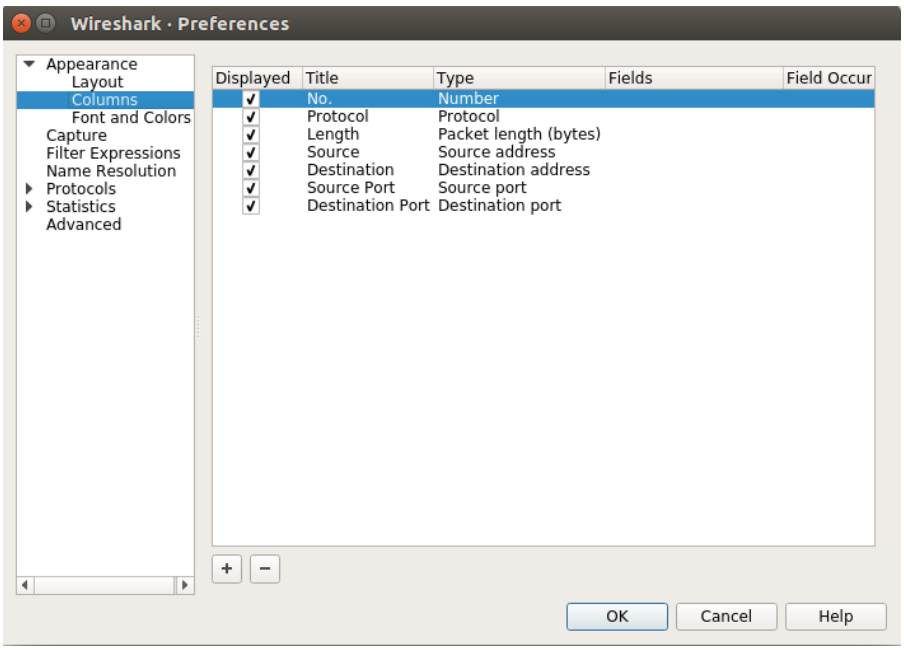
   De cada uno solo están disponibles los datos capturados por los botnets ya que los archivos de captura en su totalidad, es decir, con el trafico normal y de segundo plano se omiten por revelar información privada de los usuarios de la red.
   Se proceden a descargar los archivos de captura de los escenarios 1, 2, 4, 5, 6, 7 y 9, por cuestiones de almacenamiento disponible.
   Se renombra cada uno como "*captura-paquetes-pcap-X.pcap*", donde "X" se reemplaza por el número de captura. Luego, ejecutamos Wireshark con permisos de administrador desde la consola con "*sudo wireshark*" y abrimos la captura seleccionando la opción "Open" en el menú "*File*".

El siguiente paso se aplica a cualquiera de las dos opciones elegidas con anterioridad.

En la interfaz de *Wireshark* podemos visualizar una tabla que nos muestra las propiedades de cada paquete. Sin embargo, nosotros vamos a personalizar las propiedades visibles según nuestra conveniencia para ser utilizadas en el modelo de predicción. Para llevarlo a cabo, debemos realizar

[15] Conjunto o red de robots informáticos conectados a internet que se ejecutan de manera autónoma y automática.

clic derecho sobre una columna de la tabla e ir a la opción "*Preferencias*". Se recomienda organizar la tabla de la siguiente manera:



Cabe aclarar que el campo "*Number*" se refiere al número de paquete y solo se lo requerirá más adelante cuando combinemos estos datos con el archivo de salida de Suricata.

Luego, se procede a exportar el archivo seleccionando la opción "*Export Packet Dissections*" -> "*CSV*" en el menú "*File*" con el nombre de "*captura-paquetes-csv-X.csv*" y lo guardamos en una carpeta llamada "A-combinar".

Cada archivo de captura tiene la siguiente estructura:

```
No.","Protocol","Length","Source","Destination","Source Port","Destination Port"
"1","TCP","60","147.32.84.165","195.113.232.90","1036","80"
"2","TCP","60","147.32.84.165","195.113.232.90","1036","80"
"3","ARP","60","Cisco_db:19:c3","PcsCompu_b5:b7:19","",""
"4","ARP","60","PcsCompu_b5:b7:19","Cisco_db:19:c3","",""
"5","ARP","60","PcsCompu_b5:b7:19","Cisco_db:19:c3","",""
"6","NBNS","104","147.32.84.165","192.168.88.121","137","54053"
"7","NBNS","104","147.32.84.165","192.168.88.121","137","54053"
"8","ARP","60","Cisco_db:19:c3","PcsCompu_b5:b7:19","",""
"9","ARP","60","PcsCompu_b5:b7:19","Cisco_db:19:c3","",""
"10","ARP","60","PcsCompu_b5:b7:19","Cisco_db:19:c3","",""
"11","DNS","76","147.32.84.165","147.32.80.9","1025","53"
"12","DNS","76","147.32.84.165","147.32.80.9","1025","53"
```

Ahora debemos analizar cada archivo de captura de paquetes "*captura-paquetes-pcap-X.pcap*" con Suricata, para esto primero se tiene que "limpiar" el archivo de salida *eve.json* así nos aseguramos que la nueva salida a generar pertenezca sólo al análisis del mencionado archivo.

a) Nos dirigimos a la ubicación de *eve.json* y lo eliminamos:

```
ale@ubuntu:/var/log/suricata$ sudo rm eve.json
```

No es necesario crearlo nuevamente ya que esto se realiza de forma automática con la ejecución de Suricata.

b) Procesamos el primer archivo con Suricata añadiendo el parámetro "*-r*" para analizar un archivo de captura de paquetes en modo *offline* de la siguiente manera:

```
ale@ubuntu:~/Downloads$ sudo suricata -c /etc/suricata/suricata.yaml -r captura-paquetes-pcap-1.pcap
[sudo] password for ale:
27/11/2017 -- 09:39:16 - <Notice> - This is Suricata version 3.2.1 RELEASE
27/11/2017 -- 09:39:59 - <Notice> - all 5 packet processing threads, 4 management threads initialized, engine started.
27/11/2017 -- 09:40:04 - <Notice> - Signal Received.  Stopping engine.
27/11/2017 -- 09:40:08 - <Notice> - Pcap-file module read 323154 packets, 53096018 bytes
```

c) Y podemos verificar que el archivo de salida *eve.json* contiene el resultado de lo procesado por Suricata.

```
  GNU nano 2.5.3                                File: eve.json

{"timestamp":"2011-08-10T06:04:24.405753-0300","flow_id":1325655775523065,"pcap_cnt":22,"event_type":"dns","s
{"timestamp":"2011-08-10T06:06:36.150781-0300","flow_id":1300188775592366,"pcap_cnt":126,"event_type":"alert"
{"timestamp":"2011-08-10T06:06:36.151189-0300","flow_id":1300188775592366,"pcap_cnt":127,"event_type":"http",
{"timestamp":"2011-08-10T06:06:36.187065-0300","flow_id":1300188775592366,"pcap_cnt":131,"event_type":"alert"
```

d) Copiamos a la carpeta "A-combinar", renombramos dicho archivo como "*eve-X.json*".

Repetimos los pasos a), b), c) y d) para cada archivo de captura.

Como en este caso de opto por la opción 2), al finalizar la carpeta contendrá los siguientes archivos:

```
ale@ubuntu:~/Downloads/A-combinar$ ls
captura-paquetes-csv-1.csv  captura-paquetes-csv-6.csv  eve-2.json  eve-7.json
captura-paquetes-csv-2.csv  captura-paquetes-csv-7.csv  eve-4.json  eve-9.json
captura-paquetes-csv-4.csv  captura-paquetes-csv-9.csv  eve-5.json
captura-paquetes-csv-5.csv  eve-1.json                  eve-6.json
```

Ahora se utiliza un *script* llamado "*blender-50-50.py*" escrito en el lenguaje de programación *Python*[16] que combina cada archivo *eve-X.json* con el correspondiente *captura-paquetes-csv-X.csv*.

La función principal del mencionado script es la de crear un nuevo archivo al que llamaremos "*salida-combinacion-50.csv*". Este posee la concatenación de cada par de archivos combinados, es decir, el encabezado y los datos del archivo *captura-paquetes-csv-1.csv* sumado al campo evento junto con sus datos extraídos del archivo *eve-1.json*. Luego, la combinación resultante de los archivos *captura-paquetes-csv-2.csv* y *eve-2.json* y así sucesivamente, incluyendo solo el encabezado del primer caso por ser homogéneo.

Con el objetivo de disminuir la complejidad del árbol resultante se aplicó un filtro para limitar la cantidad de protocolos analizados, centrando nuestro análisis en aquellos que poseen más paquetes. En este caso los elegidos fueron TCP, HTTP, SSL, ICMP, DNS, IRC, TLSv1, UDP y ICMP.

Es importante destacar que luego de varias pruebas se concluyó que como la cantidad de alertas era muy pequeña en relación a la cantidad de paquetes, el árbol de decisión era muy básico. Si bien esto es un aspecto positivo ya que siempre se busca el menor árbol posible, en nuestro caso es poco ilustrativo a la hora de mostrar su funcionamiento. Para "corregir" este aspecto se implementó un filtro adicional que obtiene un 50% de paquetes de alertas por sobre el total de paquetes.

Adicionalmente, se añade un prefijo a cada línea de un archivo el cual contiene el número de archivo origen (el valor de X en su nombre). Si bien sabemos que cada línea de un archivo es única en su archivo por poseer el número de paquete, debemos asegurar que es única entre todos los

[16] Lenguaje de programación que hace foco en la sintaxis para ser interpretado fácilmente por las personas. Es multiparadigma, porque soporta orientación a objetos, programación imperativa, y en menor medida, programación funcional.

archivos porque se podría dar el caso que se tenga el mismo número de paquete en archivos distintos.

```
import json

header = True
# Este archivo es una lista de strings que contendra el resultado de la ejecucion del script
output = []
# Para cada par de archivos (captura y eve)
for x in range (1, 10):
        marked_packets = {}
        if x is not 3 and x is not 8:
                # Primero se carga el tipo de evento + el numero de dato de captura (pcap)
                with open("A-combinar/eve-"+`x`+".json", "r") as f:
                        for line in f:
                                event = json.loads(line)
                                if "pcap_cnt" in event:
                                        pcap_cnt = event["pcap_cnt"]
                                        event_type = event["event_type"]
                                        marked_packets[pcap_cnt] = event_type

                # Cargamos la salida de wireshark y la combinamos con los datos de la salida de Suricata
                with open("A-combinar/captura-paquetes-csv-"+`x`+".csv", "r") as f:
                    search_alert= True
                    first_line= True
                    for full_line in f:
                         line = full_line.strip()
                         # Si es el encabezado
                         if first_line:
                                if header:
                                        # Agregamos el tipo de evento al encabezado existente
                                        output.append(line+ ',"Event"')
                                        header = False
                                first_line = False
                         else:
                                 # Para las demas lineas
                                 columns = line.split(',')

                                 # Asumimos que la primer columna se corresponde con el numero de paquete (y eliminamos las comillas)
                                 pcap_cnt = int(columns[0].replace('"', ''))

                                 # Chequeamos si hay un tipo de evento conocido para el paquete actual y si dicho evento es una alerta
                                 if pcap_cnt in marked_packets and marked_packets[pcap_cnt] == "alert":
                                         event_type = marked_packets[pcap_cnt]
                                 else:
                                         event_type = "None"

                                 # Filtramos protocolos y agregamos un prefijo a cada linea para asegurar que sea unica
                                 # e identificar de que archivo de captura proviene
                                 if "TCP" in line or "HTTP" in line or "SSL" in line or "ICMP" in line or "DNS" in line  or "IRC" in
line or "TLSv1" in line or "UDP" in line or "SMTP" in line:
                                         if event_type == "alert":
                                                if search_alert:
                                                        output.append('%s,"%s"' % ('"00'+`x`+line[1:], event_type))
                                                        search_alert= False
                                         else:
                                                if not search_alert:
                                                        output.append('%s,"%s"' % ('"00'+`x`+line[1:], event_type))
                                                        search_alert = True

with open("salida-combinacion-50.csv", "w") as f:
            f.write("\n".join(output))
```

Nos ubicamos en la misma carpeta del script y lo ejecutamos de la siguiente manera:

```
ale@ubuntu:~/Downloads$ python blender-50-50.py
```

En dicha carpeta se creará el archivo "*salida-combinacion-50.csv*" que usaremos para el árbol de decisión.

Como se explicó con anterioridad, se aplicará una forma de aprendizaje inductivo a través de árboles de decisión para lograr que el sistema de detección de intrusos logre predecir sus acciones para con conjuntos de datos nunca antes vistos. Esto se llevará a cabo a través de una herramienta llamada *Weka*[17].

Para su ejecución debemos realizar los siguientes pasos:

Instalar la Máquina Virtual de Java (JVM).

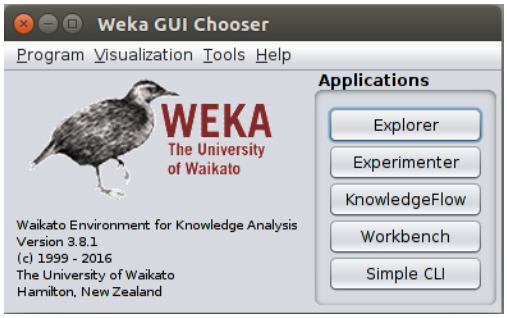
[17] *Weka (Waikato Environment for Knowledge Analysis*, en español Entorno para Análisis del Conocimiento de Waikato), es una plataforma de software libre para el aprendizaje automático y la minería de datos escrito en lenguaje *Java*. Fue desarrollado por la Universidad de Waikato ubicada en la ciudad de Hamilton, Nueva Zelanda.

```
ale@ubuntu:~/Downloads$ sudo apt-get install default-jre
```

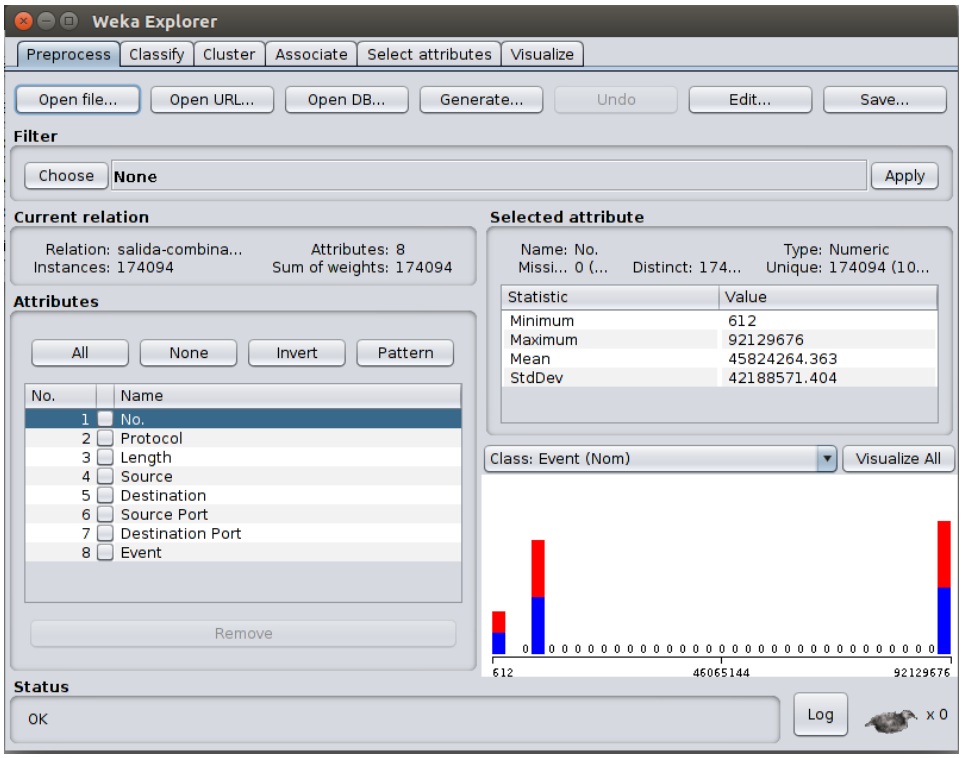
Descargamos el software Weka desde su página web, lo descomprimimos y ejecutamos con el siguiente comando:

```
ale@ubuntu:~/Downloads/weka-3-8-1$ java -jar weka.jar
```

Elegimos la aplicación “Explorer”:



En la sección de pre procesamiento abrimos el archivo *salida-combinacion-50.csv* y lo exportamos como *“salida-combinacion-50.arff”* a través de la opción “*Save…*”. El formato *arff* se utiliza para expresar los datos como una lista de instancias que comparten un conjunto de atributos.



Como paso siguiente se debe definir el conjunto de entrenamiento y el conjunto de testeo a partir del archivo anterior. Para esto, se utiliza un *script* llamado “*train&test-sets.py*” que se muestra a continuación:

```python
import random

complete = []
header = []
count = 0
 # Abro el archivo origen
with open ("salida-combinacion-50.arff","r") as source:
        first_line= True
        for full_line in source:
                line = full_line.strip()
                if count < 12:
                        # Guardo el encabezado aparte para que no forme parte de la seleccion aleatoria
                        header.append("\n"+line)
                        count+=1
                else:
                        complete.append(line)

# Tomo de forma aleatoria el 30% de las lineas del archivo origen
test_data = random.sample(complete, 52229)

# Guardamos el archivo de test
with open("salida-combinacion-50-test.arff","w") as te:
        te.write(''.join(header) + "\n")
        te.write("\n".join(test_data))

# Obtengo el archivo de entrenamiento como la diferencia del conjunto completo y la del conjunto de test
complete_set = set(complete)
test_data_set = set(test_data)
train_data = complete_set.difference(test_data_set)

# Guardamos el archivo de entrenamiento
with open("salida-combinacion-50-train.arff","w") as tr:
      tr.write(''.join(header) + "\n")
      tr.write("\n".join(train_data))
```

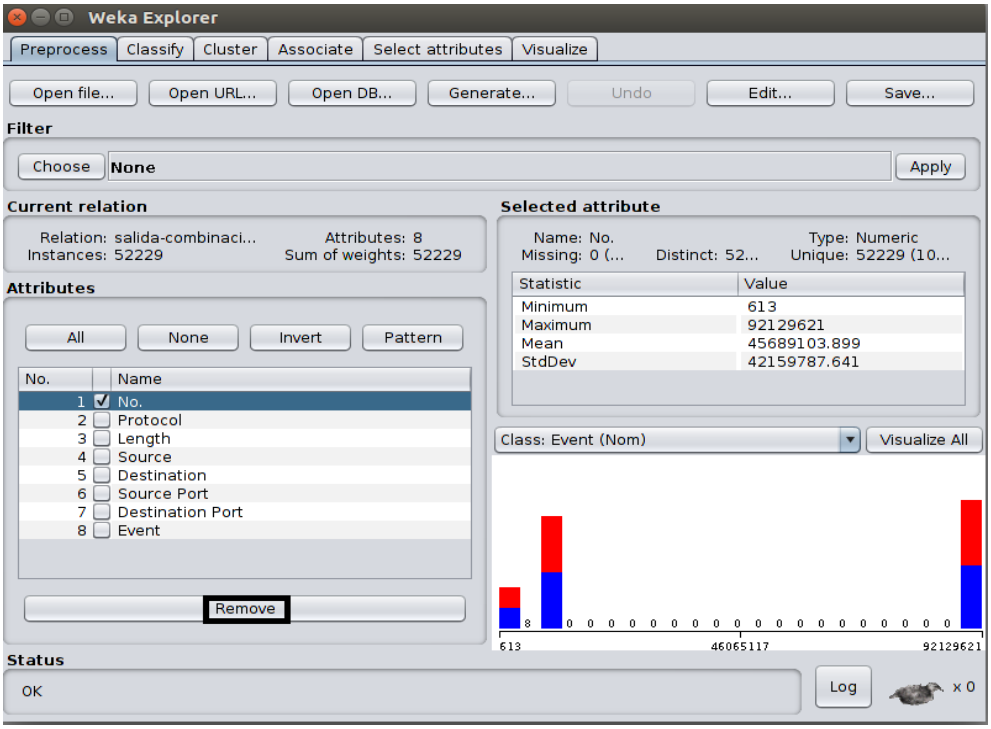
Cuya función principal es la de crear un archivo denominado "*salida-combinacion-50-test.arff*" el cual servirá para testear la predicción al final y un archivo llamado "*salida-combinacion-50-train.arff*" para crear el árbol de decisión a partir del entrenamiento.

Como primer paso, se obtiene de forma aleatoria el 30% de las líneas del archivo fuente para crear el archivo de testeo. Luego, se crea el archivo de entrenamiento con el 70% restante aplicando la diferencia entre el conjunto fuente y el conjunto de testeo para asegurar que el archivo de entrenamiento sea disjunto al de test, y así evitar el "*peeking*".

Lo ejecutamos con el siguiente comando:

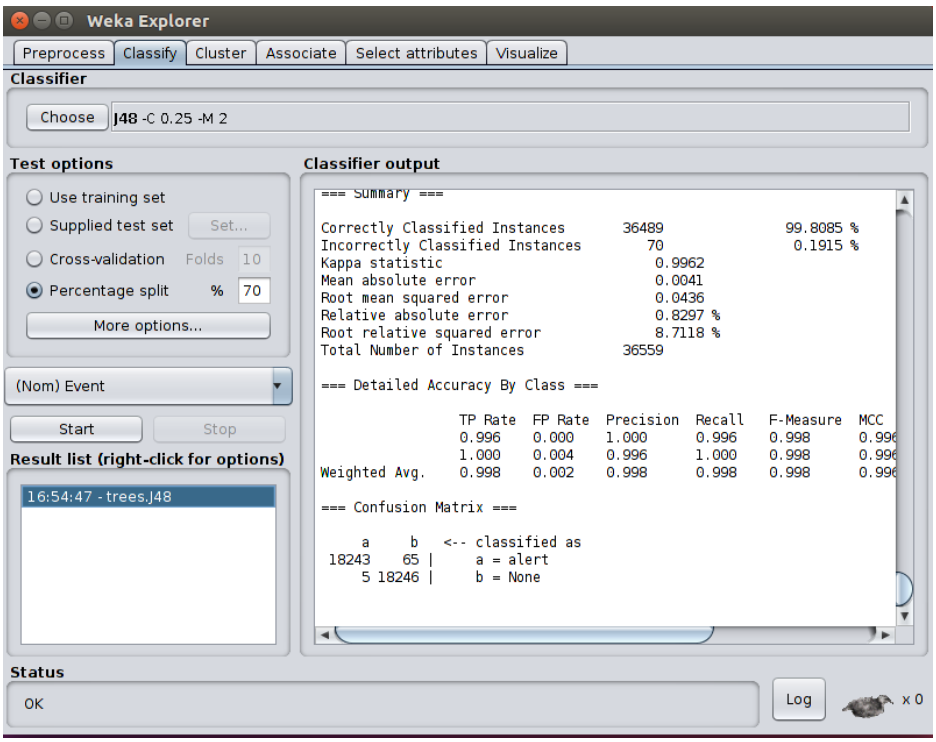
```
ale@ubuntu:~/Downloads$ python train\&test-sets.py
```

Antes de empezar con la etapa de entrenamiento debemos quitarle el atributo de numeración al archivo de test a través de Weka por no ser importante en la predicción.

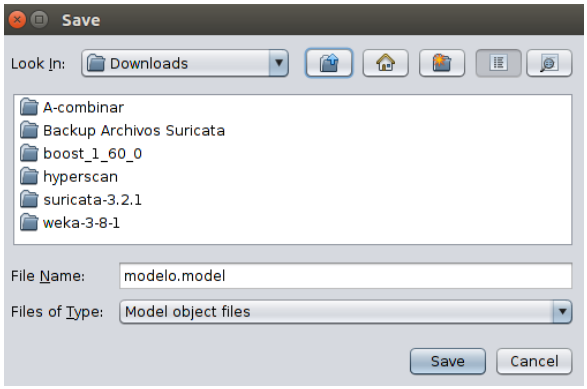


Abrimos el archivo de test con Weka, seleccionamos el atributo "*No.*", seleccionamos "*Remove*" y guardamos.

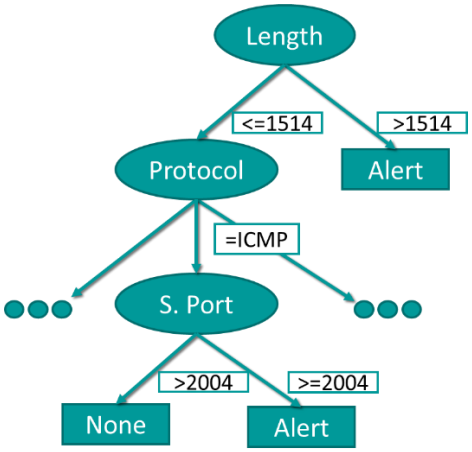
Comenzamos con el entrenamiento, para esto abrimos el archivo "*salida-combinacion-50-train.arff*" con Weka, quitamos el atributo "*No.*" como hicimos antes y nos dirigimos a la pestaña "*Classify*". Acto seguido elegimos el algoritmo usado para clasificación "J48" que será el encargado de generar el árbol de decisión. Dentro de las opciones de test seleccionamos "Percentaje split" con un valor de 70, el cual indica que porcentaje se utilizará para el entrenamiento. Verificamos que la decisión se tome en base al evento (*alert* o *None*) y damos a *Start*.



En el cuadro "*Result list*" se puede apreciar el modelo generado. Lo guardamos como "*modelo.model*" haciendo clic derecho sobre el mismo y seleccionamos "Save model". Este se utilizará en la fase siguiente.



Además, haciendo clic derecho sobre el modelo y seleccionando la opción "*Visualize tree*" podemos ver el árbol de decisión generado. Parte del mismo se muestra en la siguiente imagen:



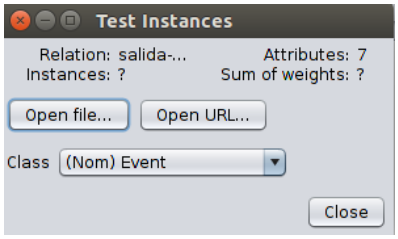
También, en la ventana de salida se puede observar el árbol podado expresado en texto plano:

```
J48 pruned tree
------------------

Length <= 1514
|   Protocol = TCP: None (60013.0/226.0)
|   Protocol = HTTP: None (261.0/16.0)
|   Protocol = HTTP/XML: None (0.0)
|   Protocol = TLSv1: None (585.0/7.0)
|   Protocol = SSLv2: None (3.0)
|   Protocol = ICMP: None (3.0/1.0)
|   Protocol = DNS
|   |   Destination Port <= 1969: None (204.0/3.0)
|   |   Destination Port > 1969
|   |   |   Destination Port <= 2078: alert (21.0/5.0)
|   |   |   Destination Port > 2078: None (19.0/1.0)
|   Protocol = SMTP: None (17.0/5.0)
|   Protocol = SSLv3: None (45.0)
|   Protocol = IRC: alert (19.0)
|   Protocol = UDP: None (26.0)
|   Protocol = SSL
|   |   Destination = 147.32.84.165: None (9.0/1.0)
|   |   Destination = 94.63.149.152: None (0.0)
|   |   Destination = 60.190.223.75: None (0.0)
|   |   Destination = 122.224.6.164: None (0.0)
|   |   Destination = 195.88.191.59: None (0.0)
|   |   Destination = 61.147.99.179: None (0.0)
|   |   Destination = 195.113.232.98: None (0.0)
|   |   Destination = 77.79.4.96: None (0.0)
|   |   Destination = 60.199.114.56: None (0.0)
|   |   Destination = 94.127.76.180: None (0.0)
|   |   Destination = 195.113.232.91: None (0.0)
|   |   Destination = 64.12.175.136: None (0.0)
|   |   Destination = 98.126.71.122: None (0.0)
```

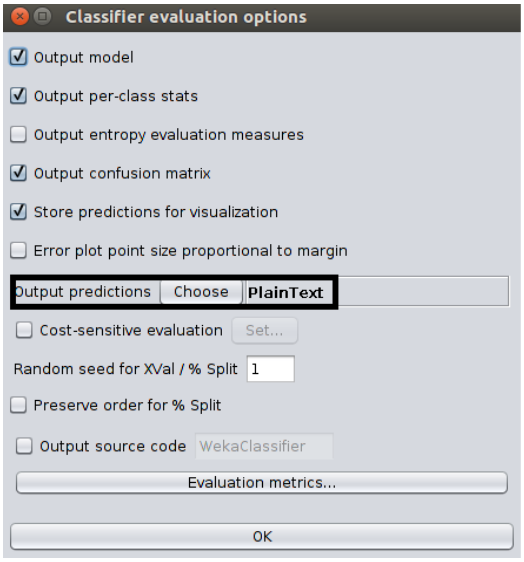
Al final de la ventana de salida se pueden ver datos como la configuración elegida, un resumen de ejecución, la precisión clasificada de acuerdo a la clase y la matriz de confusión. Esta última permite la visualización del desempeño del algoritmo, donde cada columna representa el número de predicciones de cada clase, mientras que cada fila representa a las instancias en la clase real. En este caso, de 18308 alertas reales (18243 + 65), 65 fueron clasificadas de manera errónea, en el caso de *None* 5 casos fueron mal clasificados.

```
=== Summary ===

Correctly Classified Instances       36489               99.8085 %
Incorrectly Classified Instances        70                0.1915 %
Kappa statistic                          0.9962
Mean absolute error                      0.0041
Root mean squared error                  0.0436
Relative absolute error                  0.8297 %
Root relative squared error              8.7118 %
Total Number of Instances            36559

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.996    0.000    1.000      0.996    0.998      0.996    0.998     0.998     alert
                 1.000    0.004    0.996      1.000    0.998      0.996    0.998     0.997     None
Weighted Avg.    0.998    0.002    0.998      0.998    0.998      0.996    0.998     0.998

=== Confusion Matrix ===

     a     b   <-- classified as
 18243    65 |     a = alert
     5 18246 |     b = None
```

Finalmente, se procede a realizar la predicción basada en el modelo creado, para esto abrimos el archivo fuente *salida-combinacion-50.arff* con Weka, quitamos el atributo *No.* y nos dirigimos a la pestaña *Classify*. En la sección *Test Options* seleccionamos la opción *Supplied test set* y cargamos el archivo *salida-combinacion-50-test.arff*.



Luego, debemos especificar que deseamos obtener las predicciones como resultado en formato de texto plano, para esto nos dirigimos a la opción *More options…* y modificamos el campo *Output predictions*.



Debemos cargar el modelo generado haciendo clic derecho en el campo *Result list* y seleccionando *Load model*. Por último, reevaluamos el conjunto de datos actual a través de dicho modelo haciendo clic derecho en el mismo y seleccionando la opción *Re-evaluate model on current test set*.

En la ventana que muestra la salida del clasificador podremos ver el valor de cada evento junto con su predicción. Algunas se muestran en la siguiente imagen:

```
=== Re-evaluation on test set ===

User supplied test set
Relation:     salida-combinacion-50-weka.filters.unsupervised.attribute.Remove-R1
Instances:     unknown (yet). Reading incrementally
Attributes:   7

=== Predictions on user test set ===

    inst#     actual  predicted error prediction
        1   1:alert    1:alert       1
        2   1:alert    1:alert       1
        3    2:None     2:None       0.996
        4   1:alert    1:alert       1
        5   1:alert    1:alert       1
        6    2:None     2:None       0.996
        7   1:alert    1:alert       1
        8   1:alert    1:alert       1
        9   1:alert    1:alert       1
       10    2:None     2:None       0.996
       11    2:None     2:None       0.996
       12   1:alert    1:alert       1
       13    2:None     2:None       0.996
       14    2:None     2:None       0.996
       15    2:None     2:None       0.996
       16    2:None     2:None       0.996
       17    2:None     2:None       0.996
       18   1:alert    1:alert       1
       19   1:alert    1:alert       1
       20   1:alert    1:alert       1
```

Y al final estadísticas sobre lo visto en la imagen anterior donde se aprecia un porcentaje de efectividad muy alto, 127 instancias clasificadas de forma incorrecta, la tabla de precisión con la tasa de verdaderos positivos (TP), falsos positivos (FP) y la matriz de confusión.

```
=== Summary ===

Correctly Classified Instances       52102               99.7568 %
Incorrectly Classified Instances       127                0.2432 %
Kappa statistic                          0.9951
Mean absolute error                      0.0045
Root mean squared error                  0.0488
Total Number of Instances            52229

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.995    0.000    1.000      0.995    0.998      0.995    0.998     0.998     alert
                 1.000    0.005    0.995      1.000    0.998      0.995    0.998     0.996     None
Weighted Avg.    0.998    0.002    0.998      0.998    0.998      0.995    0.998     0.997

=== Confusion Matrix ===

     a     b   <-- classified as
 25992   123 |     a = alert
     4 26110 |     b = None
```

Como consecuencia del contraste de las predicciones con la clasificación correcta se evidenciaron los buenos resultados obtenidos. Por lo tanto, se puede afirmar que el algoritmo de aprendizaje creado fue efectivo ya que sus hipótesis hicieron buenas predicciones al pronosticar clasificaciones de ejemplos que no han sido observados.

Se puede inferir que la realización de una retroalimentación a partir de los mismos impactará de forma positiva adquiriéndose la capacidad de hacer más, mejorará su desempeño y se acelerará la ejecución de las tareas. Siempre teniendo en cuenta que el sistema se utilice donde fue realizada la captura.

Capítulo 5: Conclusión

En el presente informe de proyecto final de carrera se aplicaron muchos de los conceptos vistos a lo largo de la carrera en materias como Modelos Estadísticos para Ciencias de la Computación, Sistemas Operativos, Redes y Teleprocesamiento, Sistemas Distribuidos, Seguridad en Sistemas, Conceptos de Inteligencia Artificial, entre otros. Gracias a lo anterior se pudieron introducir y detallar los diferentes tipos de Sistemas de Detección de Intrusos, ya sea clasificándolos según su enfoque o ubicación. Luego, se estudió en profundidad una herramienta en particular llamada Suricata. A partir de esto, se puso en funcionamiento dicho sistema para corroborar su correcto funcionamiento. Se realizaron pruebas adicionales utilizando reglas propias sobre una serie de malware y ataques actuales que ponen en peligro tanto a las personas como a los activos de las empresas. También, se puso énfasis a una rama de la inteligencia artificial como es la de *Machine Learning* explicando los diferentes algoritmos existentes y centrándome en los árboles de decisión.

Por último, se experimentó la utilización de los árboles de decisión a través del software Weka con el Sistema de Detección de Intrusos Suricata, para mejorar su funcionamiento gracias a la capacidad de aprendizaje. Se obtuvo un algoritmo de aprendizaje efectivo ya que sus hipótesis hicieron buenas predicciones al pronosticar clasificaciones de ejemplos nunca antes vistos. Por lo tanto, se puede afirmar que el mencionado algoritmo complementaría a dicho sistema si se aplicara una retroalimentación de los resultados del árbol de decisión expandiendo su rango de comportamiento, incrementando su precisión y rendimiento. La afirmación anterior se basa en el hecho de que a partir del resultado generado por el árbol de decisión se puede generar una expresión lógica equivalente y que toda expresión lógica se puede expresar como un condicional anidado (secuencia IF – THEN - ELSE).

Se prevé que en un futuro cercano la introducción de técnicas de inteligencia artificial a sistemas de detección de intrusos sea obligatoria debido al preocupante avance del malware. De cualquier manera, esto es una historia de nunca acabar ya que a medida que pasa el tiempo surgen nuevas metodologías de ataque que al tiempo son solucionados con nuevos sistemas de defensa, para ser burlados, y así sucesivamente.

Agradecimientos

Los artífices más importantes de este logro son mis padres Rubén y Graciela, gracias a su esfuerzo económico tuve posibilidad de estudiar una carrera universitaria y su apoyo anímico fue fundamental antes de afrontar cada parcial y final. De ellos nunca recibí negación o cuestionamiento alguno, solo confiaron en mí.

A mis hermanos Cynthia y Simón que estuvieron a mi disposición cuando más los necesite y que siempre creyeron en mí.

A mis abuelos Luis y Bocha por acompañarme en todo momento y recibirme de manera excepcional, ya sea con las deliciosas comidas de la abuela o con las salidas reconfortantes al campo con el abuelo.

A mis amigos, compañeros de estudio y de salidas. Esos con los que te juntas a estudiar días enteros a contrarreloj sin importar si es la misma materia y te contienen en tus momentos de desesperación atajando las dudas de último minuto. Y esos que te ayudan a despejar las ideas a través de una caminata, una salida al boliche y demás.

Al Director y Co-Director del presente Proyecto Final, Dr. Alejandro Javier García y Lic. Leonardo de Matteis respectivamente, por su paciencia, su sabiduría impartida y sus consejos.

A la Universidad Nacional del Sur (UNS) por ser una institución pública cuya excelencia académica permite la formación de cientos de personas año a año.

Espero algún día poder retribuir de alguna forma a todas aquellas personas que estuvieron de uno u otro modo conmigo en este largo camino, ya que sin ustedes yo no hubiera llegado hasta acá.

Gracias a todos.

Ale ☺

Referencias

- https://suricata-ids.org/
- https://es.wikipedia.org/
- http://doc.emergingthreats.net/
- https://redmine.openinfosecfoundation.org/
- https://home.regit.org
- https://01.org/hyperscan
- http://suricata.readthedocs.io
- https://security.stackexchange.com
- https://www.stamus-networks.com
- https://logentries.com/
- https://stratosphereips.org/
- https://mcfp.weebly.com
- Russell, S. & Norvig, P. (2009). Artificial Intelligence – A Modern Approach. 3rd Edition. Prentice Hall.
- P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, E. Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges.
- Alejandro J. García. (12-10-2017). "Inteligencia Artificial. Notas de Clase". Universidad Nacional del Sur.
- http://www-formal.stanford.edu/