UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA
SISTEMAS INFORMÁTICOS
CURSO 2008/2009

SISTEMA HÍBRIDO PARA LA DETECCIÓN DE CÓDIGO MALICIOSO

Jorge Argente Ferrero

Raúl García González

Javier Martínez Puentes

Dirigido por:

Luis Javier García Villalba
Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática - Universidad Complutense de Madrid

AUTORIZACIÓN

Se autoriza a la Universidad Complutense de Madrid a utilizar y difundir con fines académicos, no comerciales y mencionando expresamente a los autores, el contenido de este documento de texto, así como del contenido del CD complementario que adjuntamos con el mismo.

Jorge Argente Ferrero

Raúl García González

Javier Martínez Puentes

Dirigidos por:

Luis Javier García Villalba.

Departamento de Ingeniería del Software e Inteligencia Artificial.

Facultad de Informática. Universidad Complutense de Madrid.

AGRADECIMIENTOS

Queremos expresar nuestro agradecimiento a todos los colaboradores que han participado desinteresadamente en el desarrollo de este proyecto:

Nelson Javier Cárdenas Parra
Grupo de Análisis, Seguridad y Sistemas (GASS)
Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática. Universidad Complutense de Madrid.

Luis Javier García Villalba
Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática. Universidad Complutense de Madrid.

RESUMEN

Los Sistemas Detectores de Intrusos deben formar parte de la seguridad de todo tipo de redes y sistemas informáticos. Actualmente, la mayoría de estos sistemas de detección se basan en firmas para localizar ataques ya conocidos.

Este trabajo propone y desarrolla un prototipo de IDS híbrido que proporciona seguridad ante ataques conocidos y ante ataques nuevos. Para ello, usa un conocido IDS basado en firmas llamado Snort y propone un sistema de creación de patrones del tráfico legítimo para detectar anomalías.

El sistema está diseñado para detectar código malicioso ya que analiza todo el payload de los paquetes. Sin embargo, los resultados muestran que puede detectar otro tipo de intrusiones.

Palabras clave

Seguridad informática, Snort, firmas, anomalías, detección de intrusiones, código malicioso, sistema híbrido, patrones de comportamiento, bloomfilter, n-gram.

ABSTRACT

Intrusion Detection Systems must be included in security system of any networks or computer systems. Nowadays, the great majority of detection systems are based on signatures to find already known assaults.

This paper proposes and develops a prototype of hybrid IDS that provides safety with known and new assaults. To that effect, the system uses a known IDS based on signatures called Snort and proposes a system based on usual traffic´s patterns to detect anomalies.

This system is designed to detect malicious code since analyzes complete package´s payload. However, results show that it can detect another type of intrusions.

Keywords

computer security, Snort, signatures, anomalies, intrusions detection, malicious code, hybrid system, pattern of behaviour, bloomfilter, n-gram.

ÍNDICE

I

INTRODUCCIÓN

1. CONTEXTO DEL TRABAJO

El uso de internet y de las redes informáticas ha crecido en los últimos años de forma exponencial, tanto a nivel empresarial como a nivel doméstico. Sin embargo, este crecimiento del uso y las tecnologías han traído consigo un incremento aún mayor del número de ataques e intrusiones en los sistemas informáticos de todo el mundo. De ahí que la seguridad sea actualmente uno de los campos de investigación más importantes de la informática, sobre todo en el área empresarial.

Uno de los sistemas más conocidos e importantes para la protección de las redes y sistemas informáticos son los Sistemas Detectores de Intrusos (IDS). Los IDS analizan los eventos de un sistema o el tráfico que pasa por una red para detectar comportamiento sospechoso o anómalo y lanzar alarmas que permitan al administrador evitar o responder a los ataques.

Actualmente, los IDS más conocidos y utilizados usan detección basada en firmas, es decir comparan los datos que analizan con patrones de ataques ya conocidos. El principal problema de estos sistemas es que solo detectan ataques conocidos, un problema cada vez mayor debido al crecimiento del número de nuevos ataques y a la proliferación de programas que permiten a todo tipo de usuarios, sin grandes conocimientos, intentar acceder a un sistema de forma fraudulenta.

La idea de este proyecto surge de la empresa Tb-Secutiry, una de las empresas líderes del sector de la seguridad informática en España, que estaba interesada en desarrollar un IDS que funcionara conjuntamente con Snort, uno de los IDS basado en firmas más utilizado, pero que además detectara nuevos ataques basándose en anomalías del tráfico.

El proyecto realizado consigue aunar todas las ventajas de la detección basada en firmas, gracias a la utilización de Snort, y la detección basada en anomalías, con la creación de patrones del tráfico normal de la red. Debido a esto, supone una buena opción a la hora de proteger una red o sistema informático.

2. OBJETIVOS Y EVOLUCIÓN DEL PROYECTO

El objetivo principal del proyecto era crear un IDS, de código libre, que funcionara conjuntamente con Snort y de alguna manera correlacionara los eventos de ambos sistemas para detectar las intrusiones.

Sin embargo, antes de empezar a diseñar y desarrollar el prototipo debíamos realizar un extenso trabajo de documentación para aprender los conceptos básicos de los IDS y las técnicas que podíamos usar.

El primer objetivo marcado era la adquisición de conocimientos básicos sobre los IDS, es decir funcionamiento, tipos, técnicas, etc. Una vez analizados los principales tipos y técnicas actuales para la detección de intrusiones decidimos que nuestro prototipo debía ser un NIDS basado en anomalías para compensar la dificultad de Snort en detectar nuevos ataques.

Después de definir el tipo de IDS debíamos determinar el tipo de ataques para el cual estaría destinado. Para esto, comenzamos la documentación sobre los principales tipos de ataques y las técnicas que había para detectarlos y evitarlos. Durante este aprendizaje, observamos que la detección de código malicioso basándose en anomalías era un campo poco estudiado y con pocos sistemas desarrollados, sobretodo de código libre. Debido a esto, decidimos que nuestro NIDS basado en anomalías se centraría en la detección de código malicioso.

A continuación, la investigación se centró en las diferentes técnicas y prototipos que estaban destinados a la detección de código malicioso. Esta parte de la documentación fue la más difícil debido al poco material que había y la poca colaboración de los creadores de los principales artículos. Finalmente, se decidió implementar un prototipo basado en Payl y Anagram, dos de las propuestas más eficientes encontradas.

Una vez definido el prototipo, un NIDS basado en anomalías para la detección de código malicioso implementado con técnicas usadas en Payl y Anagram, debíamos comenzar el estudio sobre la otra parte importante del proyecto: Snort.

El estudio de Snort tuvo dos objetivos: comprender el funcionamiento del programa y contemplar la posibilidad de que nuestro prototipo fuera un preprocesador. Para lograr el primer objetivo, debíamos instalar el programa, algo nada trivial, y probarlo en alguna red, así como estudiar el funcionamiento de las reglas y las alarmas. En cuanto al segundo objetivo, la conversión de nuestro prototipo en un preprocesador de Snort ha resultado imposible debido a la poca información que dan los autores del programa sobre el tema.

Una vez acabado el proceso de documentación y definición del proyecto, se comenzó el diseño e implementación del prototipo. Se decidió usar C como lenguaje de programación debido a su bajo nivel de abstracción, lo que aportaba un gran rendimiento a la hora de procesar el tráfico.

El siguiente objetivo era probar nuestra propuesta con tráfico real. Para ello se crearon conjuntos de datos, tanto legítimo como con ataques, que se usaron para entrenar al sistema y comprobar, después, su rendimiento a la hora de detectar los diferentes ataques.

Finalmente, llegamos a la conclusión de que el objetivo principal marcado en un principio se había cumplido y habíamos desarrollado un sistema híbrido de detección de código malicioso, usando Snort como detector por firmas y, por supuesto, de código libre, para facilitar trabajos futuros a otros desarrolladores.

3. ESTRUCTURA DE LA MEMORIA

La estructura de por la memoria sigue la evolución que ha llevado el proyecto a lo largo de todo el año.

En primer lugar tenemos una pequeña introducción para explicar el contexto del trabajo y los objetivos marcados.

A continuación, se ofrece una visión general sobre los IDS. Sus características principales, los ataques que detectan, las distintas clasificaciones y tipos que hay y, por último, las diferentes propuestas para la detección de código malicioso.

Seguidamente, se expone la documentación sobre Snort. En este aparado veremos sus principales características, su arquitectura, sus modos de funcionamiento y la explicación del sistema de alarmas y reglas que usa.

En el siguiente bloque se explica el prototipo desarrollado en el proyecto, es decir, el análisis funcional, el diseño y los detalles sobre la implementación. Esta parte de la memoria será de obligada lectura para comprender el funcionamiento del programa.

A continuación, se encuentra el bloque que explica el procedimiento usado para probar el prototipo, desde la creación de los datasets hasta el análisis de los resultados obtenidos.

Por último, se exponen las conclusiones a las que se ha llegado después un año de trabajo y los trabajos futuros para mejorar el proyecto o ampliarlo.

Al final de la memoria se incluyen algunos apéndices con información útil para el lector que no tiene cabida en el resto de la memoria. Estos son un tutorial para la instalación de Snort y el manual de uso del programa desarrollado.

II

Visión general sobre los IDS

1. INTRODUCCIÓN

1.1. Definición

Un IDS o Sistema de Detección de Intrusiones es una herramienta de seguridad que intenta detectar o monitorizar los eventos ocurridos en un determinado sistema informático o red informática en busca de intentos de comprometer la seguridad de dicho sistema.

Los IDS aportan a nuestra seguridad una capacidad de prevención y de alerta anticipada ante cualquier actividad sospechosa. No están diseñados para detener un ataque, aunque sí pueden generar ciertos tipos de respuesta ante éstos.

Los IDS aumentan la seguridad de nuestro sistema, vigilan el tráfico de nuestra red, examinan los paquetes analizándolos en busca de datos sospechosos y detectan las primeras fases de cualquier ataque como pueden ser el análisis de nuestra red, barrido de puertos, etc.

1.2. Eficiencia de un IDS

Para medir la eficiencia de un IDS a la hora de detectar intrusiones en un sistema determinado se usan las siguientes características:

- **Precisión**

La precisión de un IDS es la capacidad que tiene el sistema para detectar a ataques y distinguirlos del tráfico normal de una red.

Para medir la precisión de un IDS se utilizan dos ratios: el porcentaje de falsos positivos, es decir el número de veces que se detecta un ataque que no existe, y el porcentaje de falsos negativos, es decir el número de ataques no detectados.

Cuanto menor sean estos ratios mejor será el detector, sin embargo, por lo general, cuando disminuye uno aumenta el otro por lo que lo ideal es encontrar un buen equilibrio entre los dos.

La fórmula que representa la precisión de un IDS es la siguiente:

$$Precisión = \frac{Ataques\_reales\_detectados}{Ataques\_reales\_detectados + Falsos\_positivos}$$

- **Rendimiento**

El rendimiento de un IDS es el número de eventos que es capaz de analizar un sistema. Lo ideal en este caso es que analice el tráfico de la red en tiempo real aunque esto es impracticable si se quiere analizar a fondo el contenido de los paquetes. El rendimiento de un sistema depende de la capacidad de procesamiento hardware de la que se disponga.

Cada tipo de IDS debe alcanzar un equilibrio entre la cantidad de paquetes que debe analizar y la profundidad de este análisis.

- **Completitud**

Un IDS es completo cuando es capaz de detectar todos los tipos de ataques. Por lo general, los sistemas detectan solo algunos tipos y es necesario combinar varias técnicas para conseguir la completitud.

La completitud de un sistema debe equilibrarse con la precisión, es decir, poder detectar la mayor parte de los ataques sin que el número de falsas alarmes crezca demasiado.

La fórmula que determina la completitud de un sistema es la siguiente:

$$Completitud = \frac{Ataques\_reales\_detectados}{Ataques\_reales\_detectados + Falsos\_negativos}$$

- **Tolerancia a fallos**

Es la propiedad de resistir a los ataques. Un IDS de ser seguro y robusto para evitar que un ataque lo inutilice dejando todo el sistema vulnerable.

Además de resistir a los ataques, esta propiedad determina la capacidad del IDS a resistir un fallo del sistema, por ejemplo un corte de luz o la destrucción del terminal donde se ejecuta. Es importante que un IDS guarde los patrones que utiliza para la detección y los logs que ha ido recogiendo con su ejecución.

- **Tiempo de respuesta**

El tiempo de respuesta de un IDS es el tiempo que tarda en reaccionar ante un ataque ya sea lanzando una alarma o poniendo medidas para detener o ralentizar la intrusión. Obviamente, cuanto menor sea el tiempo de respuesta más efectivo será el sistema, aunque debe asegurarse de que existe realmente un ataque antes de actuar.

1.3. Arquitectura de un IDS

Los IDS, como veremos más adelante, tienen gran cantidad tipos con características y objetivos muy distintos. Cada tipo de sistema tiene su propia arquitectura y comportamiento, sin embargo podemos generar un esquema general sobre los módulos comunes en la arquitectura de todo IDS.

La arquitectura básica de un IDS se puede ver en la figura 1.1. Debido a lo explicado anteriormente el esquema es muy simple, sin embargo se pueden ver todos los módulos principales que forman un IDS. A continuación se explica cada módulo y su función dentro del sistema:

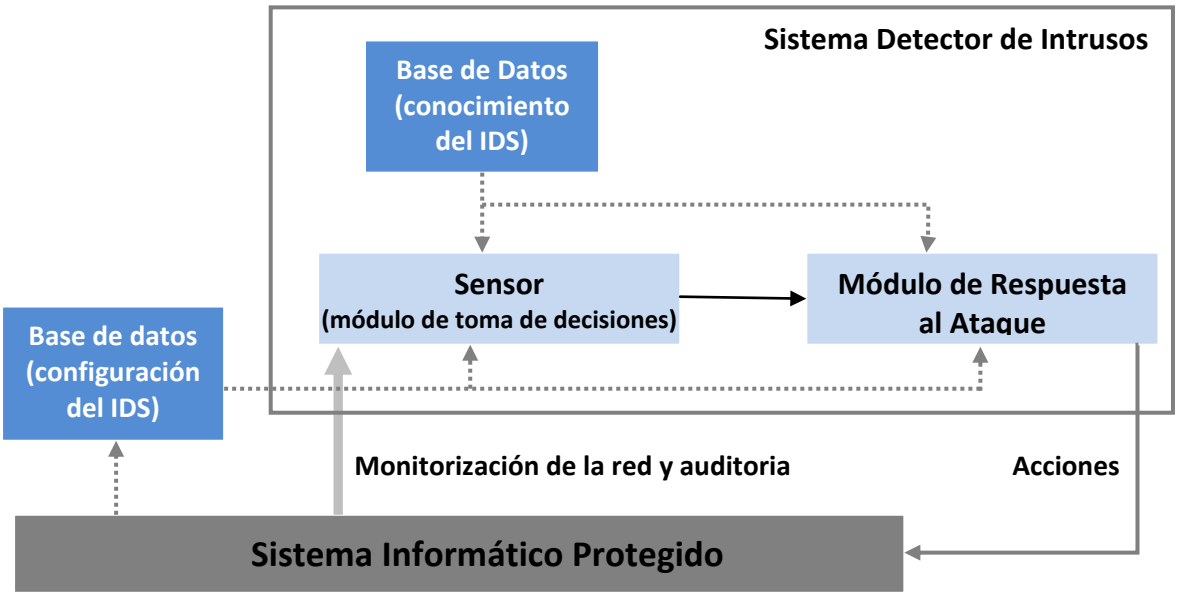

Figura 1.1. Arquitectura de un IDS

- **Sistema informático protegido**

Este módulo representa al sistema informático que queremos proteger. Su función, dentro del IDS, consiste en hacer los cambios requeridos en la base de datos de configuración y ofrecer al módulo de análisis los datos, ya sean datos que provienen de la monitorización de la red como de auditorías previas.

Después de analizar estos datos, el IDS puede realizar acciones para evitar o responder ante posibles ataques. El sistema informático será el módulo sobre el que se harán estas acciones.

- **Base de datos (configuración del IDS)**

Este módulo es una base de datos en la que se guardará toda la información referente a la configuración del IDS. La configuración vendrá, por lo general, de la mano del administrador o usuario del sistema informático a proteger ya que cada sistema suele requerir una configuración específica para que el rendimiento del IDS sea bueno.

Los datos guardados en este módulo pueden ir desde la elección del modo de funcionamiento del IDS (por ejemplo, los modos de funcionamiento de Snort) hasta parámetros propios del funcionamiento interno de los detectores (por ejemplo tipo de preprocesador usado o la activación de filtros para determinados protocolos).

Como se ve en el esquema, esta base de datos se comunica con los módulos sensor y de respuesta para determinar su funcionamiento.

- **Base de datos (conocimiento del IDS)**

La base de datos interna del IDS contiene los patrones de comportamiento que necesita para analizar los diferentes eventos del sistema informático.

Estos patrones de comportamiento cambiarán dependiendo del tipo de IDS, por ejemplo pueden definir los ataques o representar cual es el tráfico normal de la máquina protegida.

La información contenida en esta base de datos será usada por el módulo sensor y el módulo de respuesta a la hora de comparar el tráfico recibido con los patrones almacenados y determinar si existe una intrusión.

- **Sensor**

El sensor, también llamado módulo de toma de decisiones, es el encargado de analizar los datos de entrada del IDS y determinar si existe o ha existido una intrusión en el sistema que queremos proteger.

Este módulo usa el conocimiento de las dos bases de datos y algún método de clasificación o comparación para determinar si los datos analizados suponen un riesgo para el sistema. Una vez detectado un ataque manda la información al módulo de respuesta.

El sensor es la parte más importante de un IDS y, en general, está formado por varios módulos de análisis con distintos algoritmos de clasificación.

- **Módulo de respuesta al ataque**

El módulo de respuesta se encarga de realizar las acciones pertinentes, previamente definidas para cada tipo de ataque.

Cuando el sensor detecta un posible ataque manda la información sobre los paquetes sospechosos a este módulo. Aquí se comprueba el tipo de ataque que es y se busca, en las bases de datos de configuración y de conocimiento la respuesta, el tipo de respuesta que debe darse y las acciones que se realizarán sobre el sistema a proteger.

Las acciones que ordena el módulo ante los ataques van desde el lanzamiento de simples alarmas hasta la ejecución de ciertos comandos para intentar frenar el ataque o minimizar sus consecuencias.

1.4. Proceso de detección de intrusiones

El proceso que lleva a cabo un Sistema Detector de Intrusos para detectar las intrusiones en un sistema consta de 4 fases como se puede ver en la figura 1.2.

A continuación explicamos cada una de estas fases:

- **Prevención**

Lo primero que debe hacer un IDS es evitar los ataques. Para ello deberá disponer de mecanismos que dificulten las intrusiones. El sistema realizará una simulación con el tráfico que está pasando por la red o por el dispositivo y determinará si se puede llegar a una situación sospechosa, en cuyo caso se pasará a la siguiente fase del proceso.

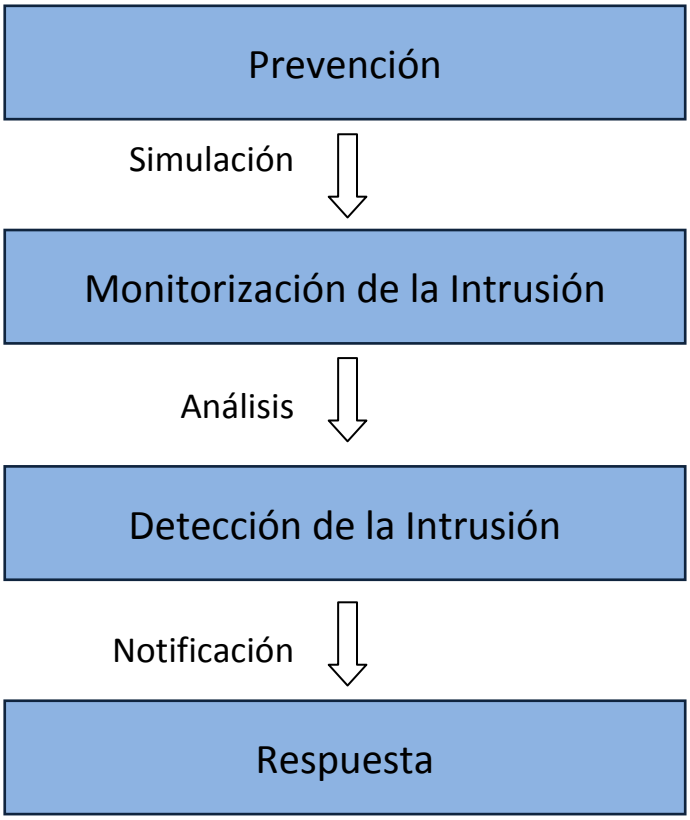

Figura 1.2. Proceso de detección de intrusiones

- **Monitorización de la intrusión**

Cuando el sistema detecte actividad sospechosa monitorizará el tráfico para que pueda ser analizado. También puede guardar un registro del tráfico sospechoso para que pueda ser revisado por el administrador del sistema más tarde. Una vez analizado el tráfico monitorizado pasamos a la siguiente fase.

- **Detección de la intrusión**

Después de analizar el tráfico, el sistema determina si la actividad sospechosa es una intrusión al sistema por medio de las diferentes técnicas que veremos más adelante. Una vez detectado el ataque deberá notificarlo. Así pasamos a la última fase del proceso.

- **Respuesta**

Por lo general los IDS no actúan para detener la amenaza o eliminarla sino que se limitan a informar al sistema o a su administrador. Asimismo crean archivos de log para poder realizar el análisis forense que permite identificar al atacante y mejorar la seguridad del sistema.

Otra forma de explicar el proceso seguido por un IDS para la detección de intrusiones es ver el funcionamiento del sistema dentro de la arquitectura.

El esquema de la figura 1.3. representa el camino de los datos desde que son recogidos hasta que se determina si pueden formar parte de un ataque.

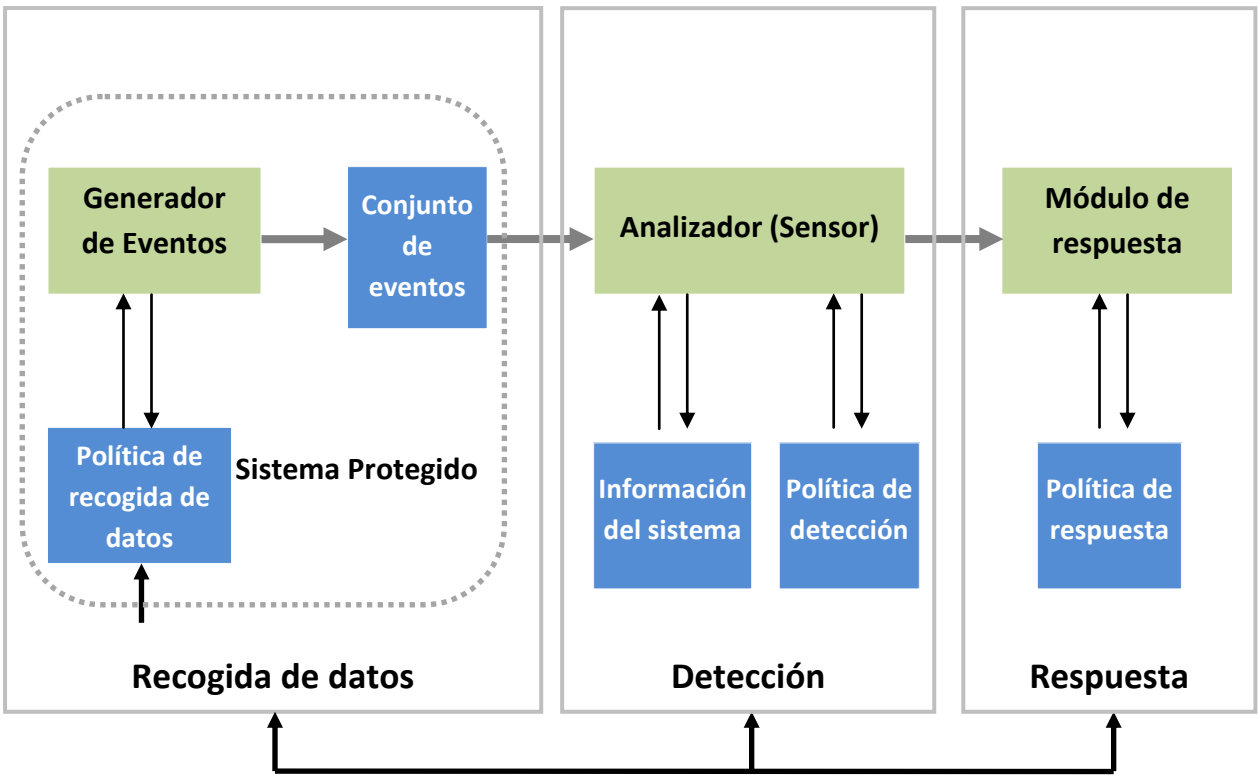

Figura 1.3. Funcionamiento de un IDS

Como se ve en el esquema hay tres zonas de funcionamiento principales.

En primer lugar, se recogen los datos siguiendo una política determinada. Esta política se refiere a la fuente y el momento en el que se

recogen los datos. Estos datos pasan al generador de eventos que creará distintos conjuntos de eventos dependiendo del tipo de datos recogidos (syslogs, paquetes de la red o información del sistema).

Estos conjuntos de eventos se usan en el módulo de detección para determinar si puede existir un ataque al sistema. Como hemos dicho antes el sensor usa la información proporcionada por el sistema y una política determinada de detección para clasificar los datos proporcionados como ataques o como tráfico normal.

Los datos generados por el módulo de detección pasan al módulo de respuesta que seguirá una política definida para responder a los ataques detectados.

2. ATAQUES

La finalidad de los IDS es detectar las intrusiones o ataques que sufre al sistema que protegen. Como hemos mencionado anteriormente, cada día surgen nuevos ataques y vulnerabilidades que complican su detección.

Según la naturaleza de los ataques estos pueden clasificarse en:

- **Pasivos**

  Se trata de intrusiones en un sistema sin consecuencias para este. Por lo general se hacen para demostrar las vulnerabilidades de un sistema o como retos que se ponen a sí mismo los *hackers* más experimentados al entrar en sistemas muy protegidos.

- **Activos**

  En esta ocasión la intrusión en el sistema se usa para dañarlo modificando o eliminando archivos. Son los ataques más perjudiciales y con consecuencias más graves.

Según el origen del atacante los ataques pueden ser:

- **Internos**

  El ataque proviene de la propia red. Puede ser realizado por usuarios de la red o por atacantes que suplantan alguna identidad. Son muy peligrosos debido a los privilegios que se tiene sobre el sistema (especialmente si trata de la identidad de un administrador).

- **Externos**

  Los ataques provienen del exterior del sistema, en general se hacen a través de internet. Son los ataques más conocidos y lo que se conoce popularmente como *hacking* o pirateo informático.

A continuación se muestra una breve clasificación sobre los ataques más frecuentes y sus posibles consecuencias en el sistema.

2.1. Monitorización

Estos ataques se ejecutan para observar a la víctima y su sistema. De esta manera, será posible obtener información muy valiosa, con el objetivo de enumerar sus debilidades y encontrar posibles formas de acceso al sistema monitorizado.

➢ Decoy

Son programas que imitan la interfaz de otras aplicaciones usadas normalmente por los usuarios. Estos programas requieren la identificación del usuario mediante su *login* y contraseña. El programa guarda la información en un fichero y ejecuta la aplicación a la que imitan para que el usuario no detecte nada anormal. El atacante podrá así suplantar la identidad del usuario del sistema.

Otros ataques de este tipo son los Keyloggers que guardan todas las teclas pulsadas por una persona en una sesión. Después, el atacante puede investigar el fichero generado para obtener contraseñas y nombres de usuarios.

➢ Escaneo

El escaneo es el método por el que se descubren canales de comunicación susceptibles de ser utilizados por el intruso. Consiste en el envío de paquetes de diferentes protocolos a los puertos de la máquina víctima para averiguar qué servicios (puertos) están abiertos según la recepción o extravío de paquetes de respuesta.

Existen diferentes tipos de escaneo según las técnicas, puertos objetivos y protocolos utilizados:

TCP connect

Forma básica de escaneo de puertos. Se intenta establecer conexión con varios puertos de la máquina objetivo. Si el puerto está escuchando,

devolverá una respuesta de éxito y se establece la conexión. Cualquier otro evento significará que el puerto está cerrado.

Esta técnica se caracteriza por ser rápida y no precisar ningún permiso de usuario sobre la máquina víctima. Sin embargo, es una técnica que se detecta fácilmente ya que los intentos de conexión queda registrados en la máquina objetivo.

TCP SYN

Este escaneo usa la técnica de "la media apertura". Consiste en mandar un paquete TCP SYN al puerto objetivo. Si este puerto está abierto contestará con un paquete ACK y si no es así responderá con un paquete RST. Si el puerto está abierto se responderá al paquete ACK con un paquete RST para no establecer la conexión y no dejar rastro en la máquina objetivo.

Es una técnica poco ruidosa y muy sutil ya que no se llega a establecer conexión en ningún momento.

TCP FIN

Consiste en el envío de un paquete FIN a un puerto. Si éste responde con un RST el puerto estará cerrado. Sin embargo, si no se obtiene ninguna respuesta significa que el puerto está abierto.

Esta técnica es la más efectiva para no ser detectados, sin embargo solo funciona en sistemas LINUX/UNIX ya que en Windows siempre responde con un RST a los paquetes FIN.

Escaneo fragmentado

Este procedimiento consiste en fragmentar los paquetes de sondeo dirigidos a la máquina víctima. Con esto conseguimos provocar menos ruido en las herramientas de protección del sistema objetivo.

- **Sniffering**

Consiste en, una vez dentro de una red de ordenadores, ejecutar en alguna de las máquinas del sistema un programa espía, llamado *sniffer*. Este programa registra en un fichero todo el tráfico que pasa por la red. En este fichero pueden descubrirse todo tipo de datos como contraseñas, números de cuenta bancaria, etc. Por lo general, estos datos estarán encriptados por lo que su lectura no será sencilla.

2.2. Validación

Este tipo de técnicas tienen como objetivo el suplantar al dueño o usuario de la máquina mediante el uso de sus cuentan de acceso y contraseñas.

- **Spoofing - Looping**

Consiste en acceder a una máquina del sistema (con una contraseña que ya tenemos) y obtener información de la red y del resto de equipos suplantando nuevas identidades de usuarios de ordenador en ordenador (siempre y cuando estén conectados a la red). Por ejemplo, si estamos en el salto 4 y realizamos un ataque de escaneo a otra máquina se registra´ra como autor del ataque al usuario del ordenador 4.

Estos ataques son muy útiles para ocultar la identidad del intruso ya que debe pasarse por todas las máquinas en las que ha entrado para encontrar el origen.

- **Suplantación de IP**

Consiste en generar paquetes de información con una dirección IP falsa. Para ello cambiamos la cabecera del paquete asignándole una IP de origen de algún usuario del sistema. De esta forma se consigue culpar a otra persona del ataque, incluso que la red de la que proviene el ataque es otra.

- **Suplantación DNS**

Consiste en la manipulación de paquetes del protocolo UDP para engañar al servidor de dominio y que éste nos entregue información de sus tablas de registros. Esto es posible solo si el servidor DNS funciona en modo recursivo, modo por defecto en los servidores de dominio.

- **IP Splicing-Hijacking**

Consiste en interceptar una conexión establecida recientemente entre un usuario y el servidor (ordenador víctima). En un momento dado, se envía al servidor un paquete casi idéntico al que espera recibir del usuario, así conseguimos suplantar su identidad. En ese momento, el intruso recibe la respuesta del servidor y puede seguir mandando paquetes, ya que el servidor cree que es el usuario el que los envía. El usuario inicial se queda esperando datos como si su conexión se hubiera colgado.

- **Puertas traseras**

Las puertas traseras son programas que se usan para entrar en un software o máquina sin pasar los procedimientos normales de validación. Si un atacante descubre una puerta trasera o consigue ejecutarla en la máquina objetivo podrá entrar en ella sin necesidad de identificarse.

- **Uso de exploits**

Los *exploits* son programas que aprovechan las vulnerabilidades conocidas de los sistemas operativos para dar al intruso la identidad de súper usuario o algún otro privilegio en la máquina víctima. Con estos privilegios el atacante puede conseguir el control del sistema.

2.3. Denegación de Sevicios (DOS)

Este tipo de ataques pretenden desbordar los recursos de la máquina objetivo de forma que se interrumpen los servicios que suministra. El objetivo es bloquear estos servicios ofrecidos por el ordenador a sus clientes.

➢ Flooding

Este tipo de ataque desborda los recursos del sistema. Consiste en saturar al ordenador víctima con mensajes que requieren establecer conexión y dar respuesta. Como la dirección IP del mensaje puede ser falsa, la máquina atacada intenta dar respuesta a la solicitud de cada mensaje saturando su buffer con información de conexiones abiertas en espera de respuesta; esto impide que pueda que oiga las solicitudes de otros usuarios que si necesitan la conexión.

Un ataque de este tipo, famoso en su tiempo pero para el cual ya están las defensas implementadas, es el *ping de la muerte.*

➢ SYN Flood

Consiste en mandar paquetes SYN a una máquina y no contestar a los paquetes ACK produciendo en la pila TCP/IP de la víctima una espera de tiempo para recibir la respuesta del atacante.

Para que este ataque sea óptimo debemos mandar los paquetes con una IP inexistente para que la víctima no pueda responder ni finalizar la conexión. Si el atacante genera miles de estas llamadas la máquina objetivo se colapsará.

➢ Connection Flood

Consiste en saturar una máquina enviándola más peticiones de conexión de las que puede soportar. Así conseguiremos que otros

usuarios no puedan acceder al servicio que ofrece. A medida que pasa el tiempo estas conexiones se van liberando, sin embargo basta con seguir mandando peticiones para mantener la máquina saturada.

- **Ataque Land**

Consiste en mandar paquetes con la dirección y puerto origen igual a la dirección y puerto destino a un puerto abierto de la máquina objetivo. Tras varios envíos de estos paquetes la máquina deja de responder. Esta técnica solo funciona en sistemas Windows.

- **Tormenta Broadcast**

Consiste en mandar una petición ICMP a una dirección *broadcast* de la red poniendo en la IP de origen del paquete la dirección de la máquina víctima. El dispositivo *broadcast* lanzará esta petición a todos los terminales de la red que responderán simultáneamente a la máquina objetivo. Todas las máquinas de la red responderán a la petición simultáneamente saturando a la víctima con respuestas que no puede procesar y produciendo un colapso en su sistema.

- **Desbordamiento NET**

Consiste en lanzar a la red, en la que se ha incursionado, una gran cantidad de paquetes sin sentido que colapsen el tráfico que circula por la red. De esta forma ralentizamos o paramos el tráfico legítimo que debe atravesar la red.

- **Obb o Supernuke**

Consiste en enviar paquetes UDP modificados, con la señal *Out of Band* activada, a los puertos de escucha del NETBIOS (137-139). La máquina los interpreta como no válidos, pasa a ser inestable y se cuelga.

Solo funciona en sistemas Windows aunque existen parches que protegen a las máquinas de este ataque.

➢ Tear drop One y Tear drop Two

Se realiza enviando paquetes fragmentados con datos que se solapen entre sí. Algunas implementaciones de colas IP no pueden ensamblar correctamente este tipo de paquetes y el ordenador se cuelga.

Por ejemplo, enviar un paquete con los datos del 1 al 100 y después enviar otro con los datos del 50 al 150 produciría el mal funcionamiento de la máquina objetivo. Este ataque funciona en diversas versiones de Unix, Linux y Windows.

2.4. Modificación

Este tipo de técnicas tienen como objetivo la modificación no autorizada de los datos y ficheros de un sistema. También pueden modificarse programas que se ejecutan en el sistema cambiando su funcionamiento.

➢ Tampering

Estos ataques están relacionados con la modificación sin autorización de datos o de los programas instalados en el sistema, incluido el borrado de ficheros. Son especialmente peligrosos cuando el atacante tiene permisos de súper usuario ya que podrá borrar cualquier información del sistema y dejarlo inservible.

➢ Borrado de huellas

Este tipo de ataque puede considerarse un complemento de otros tipos más dañinos. Consiste en modificar los ficheros log del sistema

operativo de la máquina asaltada, donde queda constancia de la intrusión, para borrar el rastro dejado por el atacante.

Cuando un intruso abandona una máquina debe borrar todo rastro de la sesión iniciada y los movimientos realizados. Esto permite no despertar sospechas en los administradores del sistema y poder realizar nuevos ataque aprovechando la misma vulnerabilidad.

- **Ataques Java Applets**

Los Applets de Java son ficheros ejecutables de programas y, en consecuencia, pueden ser modificados para que realicen acciones específicas de asalto. Java implementa grandes medidas de seguridad por lo que es difícil realizar estos ataques.

- **Ataques Java script o Visual script**

Los Java script o Visual script son capaces de iniciar aplicaciones en la máquina cliente o víctima para grabar información del disco duro, con lo cual resultan útiles para insertar virus u otros programas maliciosos en el ordenador del usuario.

- **Ataques ActiveX**

Los controles ActiveX son necesarios para le visualización de muchas páginas web. Si el usuario no dispone de ellos piden autorización para ser instalados en el sistema. Para verificar que un control ActiveX es legítimo, vienen acompañados de un certificado de autenticidad. Sin embargo, un atacante puede falsificar uno de estos certificados o, simplemente engañar al usuario, para que ejecute un programa determinado.

Un ejemplo conocido es un control ActiveX que desactivaba la petición de aceptación de los controles ActiveX de los navegadores cuando se ejecutaba. El ordenador víctima aceptaba la ejecución de todos los controles que encontraba en su navegación asumiendo que el usuario había aceptado su aceptación.

3. CLASIFICACIÓN DE LOS IDS

Los IDS se pueden clasificar de varias formas atendiendo a una serie de características. En el siguiente esquema se pueden observar estos criterios y los principales tipos de IDS.

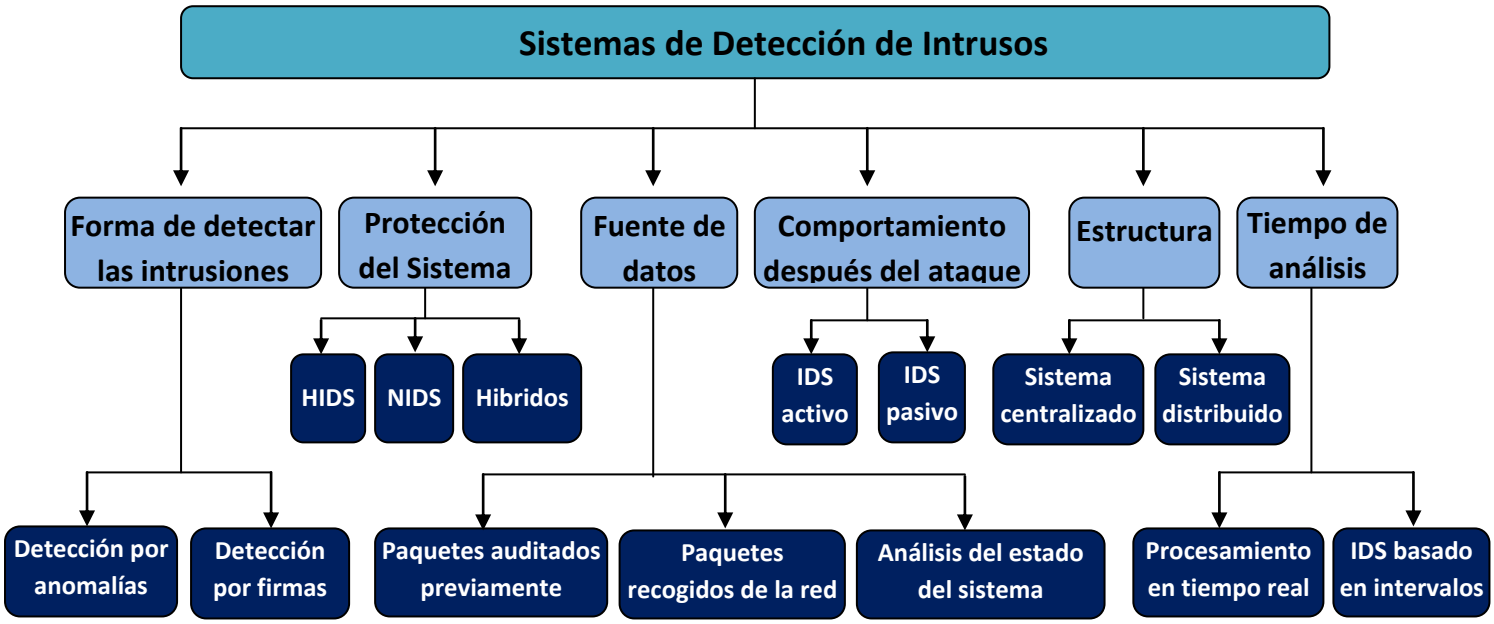

Figura 1.4. Clasificación IDS

A continuación vamos a explicar los distintos criterios y tipos y las ventajas que tienen uno respecto de otros:

3.1. Clasificación respecto a la forma de detectar las intrusiones

Esta clasificación se basa en la forma que tiene el IDS de detectar un ataque al sistema. Esto es, cómo procesa el tráfico recogido para determinar que hay una intrusión. Tenemos dos tipos:

3.1.1. Detección por anomalías

Trata de detectar intrusiones estudiando los usos anómalos del sistema. Se crean perfiles de conducta, y todo comportamiento que se sale de estos perfiles se considera un posible ataque.

El sistema tiene que ser entrenado para crear los perfiles que determinen el comportamiento normal en el sistema. Esta fase de entrenamiento debe realizarse con tráfico sin ningún tipo de ataques para garantizar el correcto funcionamiento del IDS. Los perfiles de comportamiento pueden ir evolucionando para adecuarse a los nuevos usos del sistema protegido.

La principal ventaja de estos sistemas es que pueden detectar ataques nuevos basándose en el comportamiento anómalo que ocasionen estos en el sistema. Además tienen un alto porcentaje en la detección de los ataques.

Las desventajas de este sistema son el alto número de falsos positivos ya que cualquier actividad nueva en el sistema se considerará como un ataque. La principal vulnerabilidad de este tipo de IDS es que se produzca un ataque durante la fase de entrenamiento ya que éste se considerará en un futuro como tráfico legítimo. Además, con la evolución de los perfiles un intruso que conozca el sistema puede entrenarlo para que no detecte futuros ataques. Por último decir que los IDS basados en anomalías son, por lo general, muy caros computacionalmente.

3.1.2. Detección por firmas

Estos sistemas contienen un conjunto de reglas que usarán para detectar los ataques. Su funcionamiento consiste en recoger el tráfico, analizarlo y ver si se corresponde con alguna regla en cuyo caso se identificará ese tráfico como una intrusión y se actuará en consecuencia.

El uso de estos IDS está muy extendido debido a la facilidad de su uso y a la gran cantidad de alternativas desarrolladas que hay (entre ellas Snort). Los conjuntos de reglas forman la base del sistema de detección y

de ellas dependerá la eficiencia del sistema. Por lo general, el uso de estas reglas debe configurarse correctamente para cada red con el fin de sacar todo el rendimiento al IDS.

La ventaja principal de estos sistemas es el bajo número de falsos positivos debido a que las reglas suelen ser bastante específicas. Además, pueden crearse fácilmente reglas para ataques nuevos identificados y así actualizar el detector. Por último, destacar el mínimo coste computacional que tienen estos sistemas, lo que permiten respuestas más rápidas a los ataques.

En cuanto a las desventajas de este tipo de IDS, la principal es que son pocos eficientes ante nuevos ataques, ya que si no existe una regla que identifique un ataque este no se detectará. Un problema cada vez mayor debido al gran número de ataques nuevos que surgen cada día. Esto ocasiona un gran esfuerzo para mantener el sistema actualizado mediante nuevas reglas que identifiquen estos ataques.

Por otro lado, estos sistemas pueden ser vulnerables si el atacante conoce las reglas (muchas veces de dominio público) y encuentra la forma de evitarlas. Por último decir que estos sistemas tienen problemas para detectar ataques internos al no tener en cuenta entre sus reglas las acciones desde dentro del sistema protegido.

3.2. Clasificación respecto al tipo de sistema protegido

Esta clasificación se basa en el lugar donde el IDS analizará el tráfico, es decir el tipo de protección que daremos al sistema.

3.2.1. HIDS (Host – based IDS)

Los HIDS están diseñados para monitorear, analizar y dar respuesta a la actividad de un determinado terminal (host). Su principal característica es que solo protegen el terminal en el que se ejecutan.

El funcionamiento de estos sistemas consiste en monitorizar todo tipo de actividades que sucedan en un terminal como por ejemplo: intentos de conexión, intentos de log-in, movimientos de los administradores (que tienen más privilegios y pueden ser más peligrosos), la integridad del sistema de ficheros, etc.

La principal ventaja de estos sistemas es que son los más indicados para detectar ataques internos debido a su capacidad para monitorear las acciones y accesos al sistema de ficheros que un usuario realiza en un terminal. Está demostrado que la mayoría de ataques a las empresas de hacen desde dentro, de ahí la importancia de estos sistemas de detección.

El principal inconveniente reside en que solo protegen la máquina en la que se ejecutan por lo que su uso es muy específico. Para conseguir la protección de una red es necesario usar otro tipo de sistemas.

3.2.2. NIDS (Network IDS)

Los NIDS recogen y analizan todos los paquetes que circulan por la red que tienen que proteger. Esta captura de paquetes se puede realizar desde cualquier medio y con cualquier protocolo, aunque por lo general el tráfico es TCP/IP.

Una vez recogido el tráfico se analiza de formas muy distintas para intentar detectar las intrusiones. Debido a que la captura se realiza antes de llegar a los terminales, estos sistemas suelen usarse como un primer perímetro de defensa en la protección de una red.

Las ventajas de estos IDS son que detectan los ataques antes de que lleguen a los terminales permitiendo una respuesta más rápida. Además, como hemos dicho, protegen toda la red en la que están y no solo un terminal como pasaba con los HIDS.

La desventaja de este tipo de IDS es la alta capacidad computacional que se necesita para procesar a fondo todo el tráfico que pasa por una red. En general, el análisis de los paquetes es menos profundo y detallado

que con los HIDS. Además pueden tener problemas con el tráfico encriptado, aunque este problema ya se está solucionando.

3.2.3. IDS Híbrido o NNIDS (Network-Node IDS)

Los IDS híbridos combinan los HIDS y los NIDS para obtener las ventajas de los dos sistemas. La mayoría de sistemas informáticos y redes importantes usan este tipo de detector.

Estos sistemas contienen dos módulos: uno analizará el tráfico que pasa por la red y otro estará en los terminales para analizar sus actividades. El sistema híbrido se encarga de gestionar las alertas de los dos módulos para identificar una intrusión en el sistema.

Como hemos dicho los sistemas híbridos tienen las ventajas de los dos tipos anteriores. La única desventaja es el mayor coste computacional y la dificultad para correlacionar correctamente las alertas de los dos módulos.

Dentro de los IDS híbridos tenemos los NNIDS. Estos sistemas también recoge el tráfico de la red pero después de que este haya sido dirigido hacia su terminal, es decir analiza el trafico de un solo terminal pero antes de que llegue a la máquina. La ventaja de este sistema es que puede detectar el ataque antes de que los paquetes entren en el terminal. Normalmente se usa como para te de un HIDS en los terminales críticos cuya seguridad es más importante.

3.3. Clasificación respecto a la fuente de datos

Otra forma de clasificar los IDS atiende a la procedencia de los datos que se van a analizar para detectar las intrusiones.

3.3.1. Paquetes auditados previamente

En estos IDS los paquetes provienen de un proceso de monitorización realizado previamente. Los datos están guardados en los dispositivos de almacenamiento y se analizarán después de que haya habido un ataque exitoso en el sistema. El objetivo de estos análisis es encontrar al autor del ataque y la vulnerabilidad que ha aprovechado para acceder al sistema. Esto se llama análisis forense del tráfico de un sistema.

La característica principal de estos IDS es que analizan los paquetes de forma mucho más profunda y exhaustiva que otros sistemas. Las respuestas no tienen que hallarse de forma inmediata ya que el ataque ya ha sucedido y lo importante es recoger todos los datos posibles sobre él. Aunque no se exija inmediatez en las respuestas, sí se valora la rapidez del sistema para detectar el problema de seguridad y así poder cerrar esa vía de acceso evitando futuros ataques.

3.3.2. Paquetes recogidos en la red

Este tipo de sistemas capturan el tráfico de la red actuando como un sniffer. Una vez capturado el tráfico, el sistema lo analizará para detectar posibles ataques.

Esta forma de recoger los paquetes es usada por los NIDS. Las ventajas, como hemos visto antes, es que el tráfico puede procesarse antes de llegar a su destino y, en caso de detectar una intrusión, actuar más rápidamente.

El punto negativo es que si la cantidad de tráfico es grande, el coste computacional también lo será mermando el rendimiento del IDS.

3.3.3. Análisis del estado del sistema

En este tipo de IDS no se analiza tráfico de la red sino que los datos provienen del propio sistema (kernel, servicios, archivos). Estos IDS

pueden detectar ciertos comportamientos sospechosos dentro del terminal ya sea por ser poco frecuentes o por haber sido previamente identificados como ataques. Por ejemplo, podría considerar un ataque el uso de unos archivos determinados o el acceso al sistema de un usuario desde un terminal distinto al habitual.

Como hemos visto antes, este tipo de datos son usados por los HIDS. Las ventajas son la detección de ataques internos y el poco coste computacional del procesamiento de los datos. Asimismo, el problema es que solo se protege el terminal del que se analizan los datos.

3.4. Clasificación respecto al tipo de respuesta al ataque

Los IDS también se pueden clasificar según su comportamiento después de haber detectado un ataque.

3.4.1. IDS activos

Los IDS activos, una vez detectado un ataque, crean la alarma correspondiente e intentan responder a dicho ataque para evitarlo o minimizar los daños que pueda ocasionar. Algunas de las acciones que realizan estos sistemas son cerrar puertos, aislar ciertos terminales, bloquear procesos, etc.

La ventaja de estos sistemas es que tratan de evitar el ataque automáticamente sin que el administrador del sistema tenga que analizar la alarma y actuar en consecuencia como pasa con otro tipo de sistemas.

En el lado negativo tenemos el gran crecimiento del coste computacional de estos sistemas. Al tener que responder a los ataques tienen que identificarse de forma más concreta para ser eficientes en la respuesta. Por otro lado, la dificultad de responder a un ataque de forma automática y efectiva es muy grande. Por lo general, estos sistemas

realizan acciones de prevención que no pueden tener graves consecuencias en caso de tratarse de falsas alarmas.

3.4.2. IDS pasivos

Estos IDS se limitan a generar una alerta cuando detectan un ataque. Por lo general, estas alertas contienen información sobre los paquetes que han hecho saltar una alarma y el motivo por el cual son sospechosos.

Estos sistemas son más fáciles de implementar y tienen menor coste computacional. Sin embargo, requieren la supervisión de una persona que analice las alertas y actúe en consecuencia para evitar las intrusiones o mejorar la seguridad del sistema. Por otro lado, es necesario que el sistema sea preciso a la hora de lanzar las alarmas ya que estas tienen que ser revisadas. Deben evitarse en la medida de lo posible los falsos positivos.

3.5. Clasificación respecto al tipo de estructura del sistema

Esta clasificación se basa en el tipo de estructura hardware en la que se ejecuta el IDS.

3.5.1. Sistema centralizado

Los IDS con una estructura centralizada se caracterizan por ejecutarse solo en un terminal desde el cual se controlará toda la seguridad de una red. Es el sistema usado por los NIDS que recogen todo el tráfico de una red desde un solo dispositivo.

La principal ventaja de estos sistemas es la facilidad para correlacionar eventos de distintas fuentes ya que todos los datos están en el terminal donde se ejecuta el IDS. Además, la instalación, configuración y mantenimiento es más fácil y menos costosa.

La desventaja de este tipo de IDS es la menor capacidad de cómputo con respecto a los sistemas distribuidos que aprovechan el potencial de varias máquinas para procesar los paquetes. Estos sistemas se ven limitados por la capacidad del terminal en el que se ejecutan que deberá ser grande para que el IDS tenga un buen rendimiento. Otro inconveniente es que si falla el terminal que ejecuta el detector, la red quedará desprotegida.

3.5.2. Sistema distribuido

Estos sistemas se caracterizan por ejecutarse en varios terminales a la vez ya sea por necesidad o por motivos de rendimiento. Este tipo de estructura es usado en los sistemas que usan HIDS ya que para proteger habrá que ejecutar el IDS en cada terminal que se quiera proteger. Por otro lado, también puede usarse para aprovechar la capacidad computacional de varias máquinas a la hora de analizar el tráfico recogido de una red.

Este tipo de estructura requiere un terminal que actúe como coordinador y recoja todas las alarmas generadas por los terminales satélite, las correlacione y decida si hay un ataque en el sistema.

La principal ventaja de este sistema es la mayor capacidad computacional que tiene para analizar los datos. En caso de tratarse de un HIDS esta distribución es necesaria si se quiere proteger más de un terminal.

La dificultad a la hora de recoger y relacionar correctamente todos los eventos de las distintas máquinas hace que la implementación del coordinador sea complicada. Además, aumenta el tiempo en que se detecta un ataque debido a los flujos de entrada/salida que se requieren para ello.

3.6. Clasificación respecto al tiempo de análisis

Esta clasificación se basa en la frecuencia en la que el sistema analizará el tráfico, es decir, si lo hace en tiempo real o con algunos intervalos.

3.6.1. Procesamiento en tiempo real

En estos sistemas, el análisis de los datos en busca de comportamientos sospechosos se hace en tiempo real. Por lo general, este análisis de los paquetes es bastante superficial (por ejemplo mirar solo las cabeceras de los paquetes). Para realizar un análisis más exhaustivo se tendría que reducir la capacidad de la red. Se utilizan como defensa permanente del sistema.

La ventaja de este tipo de IDS es que puede detectar los ataques en proceso y responder a ellos antes de que ocasionen grandes daños. Además, el proceso de auditoría es, por lo general, menos costoso debido a que todo el tráfico ha sido ya analizado

El problema de estos sistemas es la poca precisión en la detección debido al análisis superficial de los datos. Se debe encontrar un equilibrio entre la capacidad de la red, la capacidad de cómputo del sistema y la profundidad del análisis de los paquetes.

3.6.2. IDS basado en intervalos

Estos sistemas realizan el análisis de los paquetes a intervalos, es decir,lo solo analizan el tráfico recogido en ciertos periodos de tiempo. Es un proceso parecido al análisis forense pero con una periocidad fija.

Con estos sistemas el tráfico se analiza con algoritmos más precisos a la hora de detectar intrusiones. Son muy efectivos para detectar intrusiones recurrentes, identificar intrusiones exitosas y definir los privilegios de los usuarios. El coste computacional es menor al tener una cantidad menor de datos para procesar.

En cuanto a la parte negativa, por lo general estos sistemas detectan los ataques una vez han ocurrido por lo que solo sirven para evitarlos en el futuro. Además el primer ataque puede afectar al IDS y dejar desprotegido al sistema sin que este lo note.

4. TÉCNICAS DE DETECCIÓN DE CÓDIGO MALICIOSO

La forma más extendida de encontrar código malicioso en un sistema son los llamados antivirus. Sin embargo, estos programas solo detectan virus conocidos y necesitan constantes actualizaciones para ser eficientes contra los nuevos ataques. A continuación, vamos a conocer varias técnicas para detectar este tipo de ataques basándonos en las anomalías que provocan en el sistema respecto al comportamiento normal.

4.1. NIDS basados en payload anómalo

Estos programas detectan el código malicioso analizando el payload de los paquetes que circulan por la red. Como todos los IDS basados en anomalías tiene una fase de entrenamiento y otra de detección.

La fase de entrenamiento se hace con tráfico usado normalmente por el sistema pero completamente libre de ataques, es decir deberá ser previamente analizado en profundidad para eliminar todos los paquetes sospechosos. A continuación, se modela el payload de estos paquetes para crear patrones que identifiquen el tráfico normal y legítimo de la red. Durante la fase de detección, el tráfico a analizar se modela y se compara con estos patrones para determinar si puede ser peligroso.

Para realizar el modelado de los paquetes se usa el modelo N-Gram. Un n-gram es una secuencia de bytes adyacentes. Los paquetes se dividen en n-grams y el análisis se hace sobre un determinado número de bytes mediante una ventana de tamaño n, siendo n el número de bytes que se analizarán al mismo tiempo. Además se puede clasificar el tráfico antes de analizarlo con respecto a diferentes criterios como el número de puerto o la longitud del payload.

A continuación, se explican varias propuestas que realizan el análisis del payload para la detección del código malicioso. En general, todas son variantes de PAYL, una de las primeras propuestas que usó esta técnica con éxito.

4.1.1. PAYL

Descripción

El sistema PAYL clasifica el tráfico basándose en 3 características: el puerto, el tamaño del paquete y la dirección de flujo (entrada o salida). A continuación, crea patrones para definir lo que sería un comportamiento normal dentro de cada clase. La razón de hacer patrones basándonos en estos 3 parámetros es porque, por lo general, definen el contenido de los payloads.

Debido a la enorme cantidad de patrones que se crearán durante el entrenamiento del sistema, tendremos varios patrones por cada puerto, el sistema PAYL utiliza 1-gram para el análisis de estos patrones. La razón para usar un tamaño de n-gram tan bajo es para evitar un tamaño de memoria demasiado grande. Además, los resultados con 1-gram son bastante buenos.

Implementación y Funcionamiento

En la fase de entrenamiento el sistema genera los patrones que definirán el comportamiento normal del sistema. Como hemos dicho antes, clasifica el tráfico de entrenamiento atendiendo a su puerto destino, tamaño y dirección.

La implementación de los patrones se hace con dos arrays de 256 posiciones por cada posible clase (puerto - dirección -rango de tamaño). Para cada clase de paquetes se analiza la frecuencia en que aparecen cada uno de los 256 bytes y se construye un histograma con la media de las frecuencias y otro con la desviación típica para cada byte. En un array se almacenarán las medias de las frecuencias de cada byte y en el otro las desviaciones típicas.

Con cada nueva muestra de la fase de entrenamiento se actualizará el patrón. Para esto, no es necesario tener todos los datos anteriores ya que tanto la nueva media como la nueva desviación típica se calculan partir de la media o desviación típica actual y la nueva muestra usando las siguientes ecuaciones:

$$\bar{x} = \frac{\bar{x} \times N + x_{N+1}}{N+1} = \bar{x} + \frac{x_{N+1} - \bar{x}}{N+1} \qquad Var(X) = E(X - EX)^2 = E(X^2) - (EX)^2$$

Según se van creando los histogramas se van procesando las muestras analizadas. Si una muestra difiere demasiado del patrón que se está construyendo se desecha. Esto hace al sistema tolerante a pequeños ataques dentro del tráfico de entrenamiento ya que serán eliminados. Por otro lado, las muestras con histogramas parecidos y tamaños de paquete similares se fusionan (*clustering*) para reducir la cantidad de memoria necesaria.

Para realizar el *clustering* se sigue el siguiente proceso. En primer lugar, se fusionan aquellos perfiles cuyo tamaño de paquete es muy similar (se determina un umbral para considerar dos tamaños similares). Después se fusionan los perfiles similares en cuanto a la distribución de su histograma. Para calcular esta similitud se calcula la *distancia de Manhattan* de la muestra y se determina un umbral.

La fase de entrenamiento es no supervisada, es decir que la máquina realiza todo por sí misma sin necesidad de intervención humana.

En la fase de detección, el tráfico de la red se compara con los patrones generados durante el entrenamiento y se clasifica como legítimo o malicioso. Además, durante esta fase, dichos patrones se van actualizando a medida que el tráfico de la red evoluciona. Puede ponerse un parámetro para dar más importancia a los cambios o mantener los patrones iniciales.

En la fase de detección se calcula el número de veces que aparece cada byte dentro del paquete a analizar y se compara con la media del patrón adecuado. Para ello se usa la distancia reducida de Mahalanobis que tiene en cuenta la desviación típica (se usa la versión reducida para reducir el coste computacional). Si la distancia calculada es mayor que un cierto umbral se generará la alerta correspondiente. La fórmula para calcular la distancia de Mahalanobis es la siguiente:

$$d(x, \bar{y}) = \sum_{i=0}^{n-1} (| x_i - \bar{y_i} | / (\bar{\sigma_i} + \alpha))$$

4.1.2. Poseidon

Esta técnica se desarrolló para corregir los errores que surgen en la construcción de modelos en Payl cuando se aplica el *clustering* sobre el tamaño de los paquetes.

Descripción

El funcionamiento principal del sistema es como el de Payl. La principal diferencia es el uso de mapas auto organizativos (SOM) para clasificar el tráfico. Los paquetes son preprocesados por medio del SOM que da como salida las diferentes clases en que deben clasificarse los paquetes. En este modelo, en lugar de clasificar los paquetes atendiendo al puerto, dirección y tamaño del paquete, se clasifican teniendo en cuenta el puerto la dirección y las clases devueltas por SOM. Con esto se consigue reducir en gran número el número de clases.

El uso del SOM lleva consigo la necesidad de entrenarlo a parte del resto del sistema. Sin embargo, el beneficio que se obtiene al reducir el número de clases tanto en espacio como en precisión compensa este esfuerzo.

Funcionamiento del mapa auto organizativo (SOM)

La red está compuesta por múltiples nodos, cada uno con un vector de pesos asociado. Este vector deberá ser tan grande como el tamaño del mayor dato de entrada a procesar. El proceso de clasificación de SOM tiene 3 fases:

- **Inicialización**

  En primer lugar se deben determinar ciertos parámetros fijos del SOM como el número de nodos, la tasa de aprendizaje o el radio. El número de nodos determinará directamente el número de clases que se obtendrán al analizar el tráfico, cuanto mayor sea el número de nodos mayor será el número de clases. Tras esto, el vector de pesos se inicializa con valores en el rango de los datos de entrada.

- **Entrenamiento**

  El entrenamiento del sistema se hace comparando cada dato de entrada (los fragmentos de los paquetes) con todos los vectores de pesos mediante una función de distancia. Después se determina el nodo al que pertenecería el dato, que será el más cercano en la comparación. A continuación, se actualizan los vectores de peso de todos los nodos cercanos a la muestra (todos los que estén dentro de un radio determinado), incluido el nodo al que pertenecerá el dato.

  Gráficamente se podría ver como que los nodos están distribuidos en el espacio cartesiano (el vector de pesos serían las coordenadas) y con cada nueva muestra (la cual también tendrá sus coordenadas) los nodos más cercanos a ella se ven atraídos modificando su posición (coordenadas = vector de pesos) y acercándose a las coordenadas de la muestra.

  Este proceso se repite con todos los datos que necesita analizar el sistema. A medida que avanza el entrenamiento los vectores de peso de los nodos se van estabilizando y la tasa de aprendizaje y el radio irán disminuyendo hasta ser 0.

- **Clasificación**

  Durante la clasificación se lleva a cabo el mismo proceso que en el entrenamiento. Cada muestra es comparada con todos los nodos mediante una función de distancia, y se determina el nodo (la clase) a la que pertenece, que será el más cercano. Sin embargo, los pesos ya no se actualizan, es decir los nodos creados en la fase de entrenamiento permanecen estables.

Pruebas y resultados

Para realizar las pruebas se usó el benchmark de DARPA de 1999. Los resultados obtenidos y su comparación con Payl se pueden ver en la tabla 1.1.

		PAYL	POSEIDON
Number of models used		4065	1622
HTTP	DR	89,00%	100,00%
	FP	0,17%	0,0016%
FTP	DR	95,50%	100,00%
	FP	1,23%	0,93%
Telnet	DR	54,17%	95,12%
	FP	4,71%	6,72%
SMTP	DR	78,57%	100,00%
	FP	3,08%	3,69%
Overall DR with FP < 1%		58,8% (57/97)	73,2% (71/97)

Tabla 1.1. Comparación entre Payl y Poseidon
(DR = Ratio de detección, FP = Falsos Positivos)

4.1.3. Combinación de múltiples clasificadores de una clase

Esta técnica, también basada en Payl, fue desarrollada para eliminar la vulnerabilidad del sistema original ante los ataques mimicry.

Descripción

El sistema tiene varios detectores SVM, es decir detectores de una sola clase. La precisión del sistema aumenta a medida que aumente el número de detectores.

Los detectores de una clase solo distinguen dos posibilidades: pertenencia o no a la clase. El entrenamiento de cada detector solo se hace con muestras de la clase que va a detectar. De ahí la necesidad de tener varios detectores para abarcar varias clases distintas.

Funcionamiento

En cuanto al funcionamiento el sistema es similar a Payl. Sin embargo, este sistema era vulnerable a los ataques *mimicry* ya el uso de 1-gram para crear los modelos hacían que el atacante solo tuviera que evadir un modelo para entrar en el sistema.

Para evitar este problema, se propone el uso de varios modelos para cada paquete. Estos modelos están basados en 2-gram pero con un parámetro (v) que indicará la distancia en bytes de los 2n-gram seleccionados (aquí se llamarán 2v-gram). De esta forma, no se tienen en

cuenta todos los bytes del paquete y el modelo es más aleatorio y ocupa menos en memoria. Una vez se han conseguido los modelos, el sistema funciona como Payl, es decir comparando el tráfico a analizar con los modelos mediante la distancia reducida de Mahalanobis. Como existen varios modelos distintos en el sistema (que se crean variando el parámetro v), un ataque deberá evadirlos todos para tener éxito, lo cual complica la intrusión.

Esta técnica funciona como si se tuvieran varios sistemas Payl de 2-ngram funcionando simultáneamente pero con menos gasto de memoria, ya que cada sistema solo tendría algunos bytes de los paquetes. Durante la fase de detección, se compara cada paquete con las diferentes representaciones del patrón y si no encaja en alguna se lanzará la alarma correspondiente.

Usar esta técnica con v = 0..N, es decir, un espacio con información de 2-gram otro con 2(1)-gram,...,2(N)-gram sería equivalente a hacer el estudio con un único espacio de (N+2)-grams.

Implementación

En cuanto a la implementación se usan dos arrays de bits por cada modelo, uno con las medias y otros con las desviaciones típicas. Como se usa 2-gram los arrays deberían ser de 256*256 posiciones lo cual supondría un gasto enorme de memoria si el número de modelos es suficientemente grande. Para evitar esto, utiliza la técnica de *clustering* fusionando los conjunto 2-gram similares. Para fusionar los 2v-gram, estos tienen que tener el mismo valor v y una frecuencia de bytes parecida.

Pruebas y resultados

Para realizar las pruebas y comparar el rendimiento con el sistema Payl, ambos sistemas fueron entrenados con tráfico legitimo y ejecutados durante 4 días analizando tráfico normal y tráfico con ataques polimórficos (que intentaban ataques mimicry).

Los resultados obtenidos en los dos sistemas se pueden ver en las siguientes tablas:

DFP(%)	RFP(%)	Detected attacks	DR(%)
0.0	0.00022	1	0.8
0.01	0.01451	4	17.5
0.1	0.15275	17	69.1
1.0	0.92694	17	72.2
2.0	1.86263	17	72.2
5.0	5.69681	18	73.8
10.0	11.05049	18	78.6

Tabla 1.2. Resultados con 1-gram Payl

DFP(%)	RFP(%)	Detected attacks	DR(%)
0.0	0.0	0	0
0.01	0.00381	17	68.5
0.1	0.07460	17	79.0
1.0	0.49102	18	99.2
2.0	1.14952	18	99.2
5.0	3.47902	18	99.2
10.0	7.50843	18	100

Tabla 1.3. Resultados con 40 detectores SVM

Analizando los resultados, observamos como en Payl debemos asumir una tasa de falsos positivos (DFP) de un 5% para detectar todos los ataques, mientras que con SVM la tasa será de 0.5%. En general, se observa que para la detección de estos ataques SVM funciona mucho mejor.

4.1.4. Análisis incompleto del payload

Esta técnica surge para corregir el defecto de Payl de no poder procesar los paquetes grandes en las redes rápidas con la suficiente velocidad.

Descripción

La característica principal de este sistema es que no analiza todo el payload de los paquetes si no que los divide y reduce para realizar los patrones y las comparaciones. De esta forma se consigue procesar todo el tráfico en redes rápidas.

Para la creación de los patrones y la posterior comparación del tráfico se usa el mismo sistema que en Payl es decir se usa 1-gram y se guarda la frecuencia de aparición de los distintos bytes de cada clase. Se realiza un patrón para cada servicio que tiene el terminal, por lo cual se debe entrenar el sistema en cada máquina en que se ejecute.

Para dividir el payload se usa el algoritmo de particionado CPP, cuyo funcionamiento explicamos a continuación. El tráfico se va procesando

con una ventana de 1-gram. Para cada 1-gram se ejecuta el *rabin fingerprint*, es decir se obtiene un número entero que lo representa. A continuación, se mira si este número cumple una condición asignada previamente (por ejemplo que el módulo 1000 del número esté entre 100 y 200). Los 1-gram cuyos *fingerprints* cumplan dicha condición serán los que delimiten las particiones del payload que se analizará.

Debido a que se trata de un proceso estadístico, es posible que se creen particiones demasiado pequeñas, de forma que su distribución de bytes no sea representativa. Para solucionar esto se impone un tamaño mínimo de partición.

A la hora de construir los perfiles de cada servicio se define un parámetro n que determinará en número de particiones del payolad que se analizarán (se procesarán las n primeras particiones de cada paquete). Con un valor adecuado de n, el perfil obtenido mediante el análisis de las particiones es muy similar al obtenido si se hubiera utilizado todo el payload.

Durante la fase detección, los paquetes también son divididos mediante el algoritmo CPP y se usarán sus n primeras particiones para realizar las comparaciones con los perfiles, teniendo n el mismo valor que en la fase de entrenamiento. Para realizar las comparaciones de los perfiles se usa la distancia reducida de Mahalanobis, al igual que en Payl.

Implementación

La implementación de los patrones es igual que en Payl. La única diferencia del algoritmo es que a medida que se procesa el payload de un paquete van haciendo las particiones. Una vez que se ha llegado a n particiones dentro de un paquete se deja de analizar ya que ya tenemos la información que necesitamos.

Pruebas y resultados

El sistema se probó con el dataset DARPA 1999. A continuación se pueden ver los resultados obtenidos por Payl y los resultados usando CPP:

	Payload profiling using full payload	Payload profiling using CPP
Attacks detected	61	60
Average % of payload considered	100	61.13
Average % of false positive rate	0.0623	0.1407
Time required to process 100 packets in the training phase	102.592 ms	101.269 ms
Time required to process 1000 packets in the testing phase	5.171 ms	5.014 ms

Tabla 1.4. Comparación entre Payl y CPP

Puede observarse como la precisión es algo menor en este sistema, sin embargo el tiempo necesario para procesar el tráfico también es menor.

4.1.5. Anagram

Esta técnica es una evolución de Payl desarrollada por los mismos autores para corregir las deficiencias que tenía el sistema original. Las principales ventajas de Anagram con respecto a Payl son:

- Mayor precisión para detectar ataques, sobre todo los mimicry.
- Eficiente computacionalmente debido al uso de funciones hash en los bloom filters.
- Eficiente en uso de memoria debido al uso de bloom filters.
- Correlación rápida entre distintos sistemas ya que puede cambiar la información de los bloom filters sin comprometer la privacidad de los datos.
- Creación de firmas de nuevos ataques usando la correlación entre distintos sistemas.

Descripción

Al igual que en Payl el sistema se basa en n-grams para procesar los paquetes y crear los patrones de comportamiento. Sin embargo, usa *bloom filters* para poder dividir los paquetes en n-grams de tamaños mayores que 1 sin que el coste en espacio y rendimiento del sistema se vea perjudicado.

El *bloom filter* es una estructura para almacenar de forma binaria la aparición o no de un determinado n-gram pero no el número de veces que aparece, es decir, en este caso no almacenaremos la frecuencia de distribución de cada byte sino su aparición dentro de los paquetes.

Durante la fase de entrenamiento se rellenan dos *bloom filters*: uno con el tráfico normal y legítimo de la red, y otro con los n-grams que aparecen en virus conocidos. El proceso para rellenar esta última estructura debe ser supervisado debido a la utilización de ataques. Es por esto que se dice que el entrenamiento del sistema es semi-supervisado.

Durante la fase de detección se obtienen los n-grams de cada paquete y se comprueba si aparecen en alguno de los dos *bloom filters*. A continuación, se asigna una puntuación a este paquete teniendo en cuenta los n-gram que no aparecen en el patrón de tráfico legítimo y los que aparecen en el patrón del código malicioso. Si esta puntuación supera un determina umbral el paquete se considerará un ataque.

El uso de n-grams más grandes permite una mayor precisión ya que se tienen en cuenta las dependencias entre bytes. Por otro lado, al no tener en cuenta la frecuencia de aparición de los n-gram pueden detectarse ataques ocultos entre tráfico legítimo.

Bloom Filters

Un bloom filter se trata, básicamente, de un array de bits en el que cada posición representa a un n-gram. Cada bit del array indicará si el n-gram al que representa se encontraba en el tráfico de entrenamiento.

Como el tráfico de entrenamiento debe ser grande y los n-grams usados también, el número de posiciones que se necesiten puede llegar a ser inabarcable. Debido a esto, se utiliza una función hash universal para resumir los n-grams y limitar el número de posibilidades. Esta función hash debe garantizar las mínimas colisiones posibles entre elementos estadísticamente independientes.

El funcionamiento de este sistema es el siguiente: cuando se procesa un n-gram se hacer la función hash de sus bits obteniéndose una cadena de bits que representarán una posición dentro del bloom filter. Debido a las posibles colisiones, se suelen utilizar más de una función hash de forma simultánea de modo que para cada elemento le correspondan dos posiciones distintas del array. De esta forma si al menos una de las posiciones del array que le corresponde a un determinado elemento tiene el valor 0 querrá decir que ese n-gram no ha sido detectado previamente.

La principal ventaja del uso del *bloom filter* es el ahorro en espacio que supone a la hora de crear los patrones. Además, como el coste de las operaciones en esta estructura es de O(1) el rendimiento computacional también es muy bueno.

Una decisión importante a la hora de diseñar el sistema será el tamaño del *bloom filter* ya que este no podrá ser redimensionado posteriormente (la función hash no serviría) sin perder los datos almacenados.

Aprendizaje semi-supervisado

Anagram no es resistente a fallos durante el entrenamiento ya que si un paquete malicioso se cuela en el tráfico limpio se considerará legítimo en el futuro. Para evitar esto se crea el segundo *bloom filter* con los datos de ataques conocidos lo que se conoce como entrenamiento semi-supervisado.

Durante la fase de entrenamiento, el tráfico se comparará con el *patrón* de ataques antes de procesarse para completar el otro patrón. Si un paquete tiene un n-gram que está en el *bloom filter* de ataques este se desecha. Si el porcentaje de n-grams sospechosos de un paquete supera el 5 % se desecha todo el paquete.

A medida que avanza el entrenamiento, el contenido del *bloom filter* se irá estabilizando al encontrarse cada vez con menos n-grams nuevos. Se debe determinar un tiempo de entrenamiento mínimo que permita al patrón estabilizarse lo suficiente para representar correctamente al tráfico legítimo. Hay que tener en cuenta que cuanto mayor sea el tamaño de los n-grams, mayor será el tiempo necesario para crear un buen modelo.

Durante la fase de detección se usarán los dos modelos para asignar una puntuación a cada paquete y determinar si puede ser un ataque. Obviamente, si un n-gram aparece en el patrón de ataques tendrá más posibilidad de ser peligroso que si no aparece en el patrón del tráfico legítimo.

El entrenamiento semi-supervisado permite una mayor efectividad a la hora de evitar los falsos positivos como se puede ver en la siguiente gráfica:

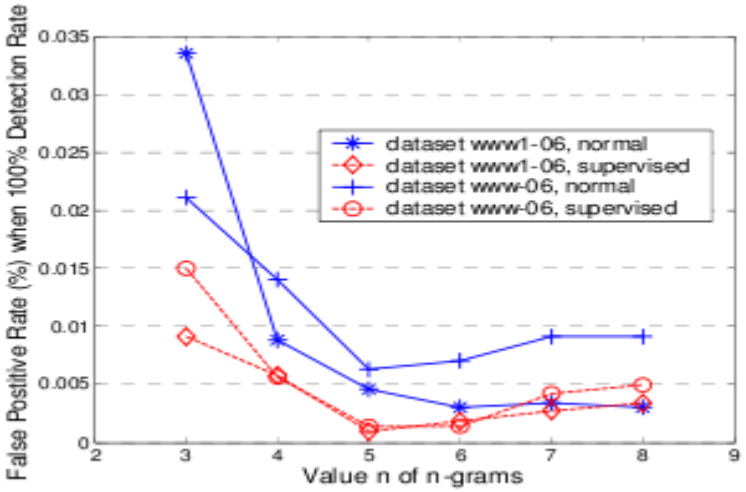

Figura 1.5. Comparación entre entrenamiento semi-supervisado y entrenamiento normal

Aleatorización

Para aumentar la resistencia del sistema frente a ataques mimmicry se sigue una técnica de aleatorización de los patrones. Esta técnica consiste en analizar el payload en dos fragmentos. Esta partición es aleatoria y se consigue por medio de la aplicación de una máscara binaria sobre el payload.

La máscara usada para particionar el payload debe ser aleatoria. Además debe crear fragmentos equilibrados y lo suficientemente grandes para que se puedan extraer n-grams de ellos.

Implementación

Para implementar los patrones se usarán dos arrays de bits de tamaño fijo N. Uno se utilizará para registrar n-grams legítimos y otro para registrar n-grams maliciosos. A la hora de determinar el tamaño de los arrays se debe alcanzar un equilibrio entre la memoria disponible y el número de colisiones.

Durante el entrenamiento se rellenará el *bloom filter* poniendo un 1 en la posición de los arrays que representen el n-gram procesado.

4.2. Búsqueda del gen de autoreplicamiento

Introducción

Una de las características que definen a la mayoría de programas maliciosos es que intentan autoreplicarse para expandirse por la red. Los métodos para que un programa pueda reproducirse son conocidos y relativamente pocos (alrededor de 50). Esta técnica analiza el código de los programas buscando estos métodos (gen de autoreplicamiento) para detectar código malicioso.

La principal dificultad es que las instrucciones de autoreplicación pueden estar ocultas entre el resto de instrucciones, o aún peor estar encriptadas. Debido a esto la búsqueda del "gen de autoreplicación" puede entenderse como la búsqueda de una serie de palabras dentro de un array de letras teniendo en cuenta los siguientes puntos:

1. Todas las palabras se posicionan a lo largo de un string y deben leerse de izquierda a derecha, en orden de ejecución.

2. Las palabras que estamos buscando pueden estar fragmentadas. Por ejemplo, si buscamos la palabra REPLICACION, ésta puede estar dividida en dos partes: RE y PLICACION. Esto implica que una vez encontrada la primera letra (R) debemos ser capaces de decir si con el resto de letras se puede completar la palabra. Para ello existen diversas técnicas de desencriptación.

3. El código malicioso puede llegar parcialmente codificado y decodificarse en un orden determinado. Por ello, durante el proceso de decodificación se tienen que hacer diversas interrupciones para analizar la composición de la imagen del ejecutable.

El gen de autoreplicación (SRG)

A continuación podemos ver un ejemplo típico de SRG.

```
Offset          Opcode              Instruction
.text:00401A57  E800000000          call  $+5
.text:00401A5C  loc_401A5C:
.text:00401A5C  5B                  pop  ebx
.text:00401A5D  81 EB 5C 1A 40 00   sub  ebx, offset loc_401A5C
```

Figura 1.6. Gen de autoreplicamiento

De una manera más formal la estructura de un SRG puede representarse usando una gramática: **G = {V_n, V_t, P, S}** siendo,

G el gen de autorreplicación

V_n la variable no terminal (corresponde a los *upper bloks*)

V_t la variable terminal (corresponde a los *lower bloks*)

P el conjunto finito de reglas de la forma: $\alpha \rightarrow \beta$ (α,β pertenecen a V_n)

S el punto de inicio del gen.

El vocabulario de la gramática V(G) está compuesto por V_n y V_t:

$$V_N = \left\{ \begin{array}{l} \text{<Self\_Relocation\_Gene>, <Call\_Command\_Block>,} \\ \text{<Pop\_Out\_of\_Stack\_Block>, <Move\_Data\_Block>,} \\ \text{<Exchange\_Data\_Block>, <Subtract\_Data\_Block>} \end{array} \right\}$$

$$V_T = \{\text{CALL, JMP, POP, MOV, LEA, XCHG, SUB, ADD}\}$$

Esta idea sigue el siguiente diagrama de la figura 1.7.

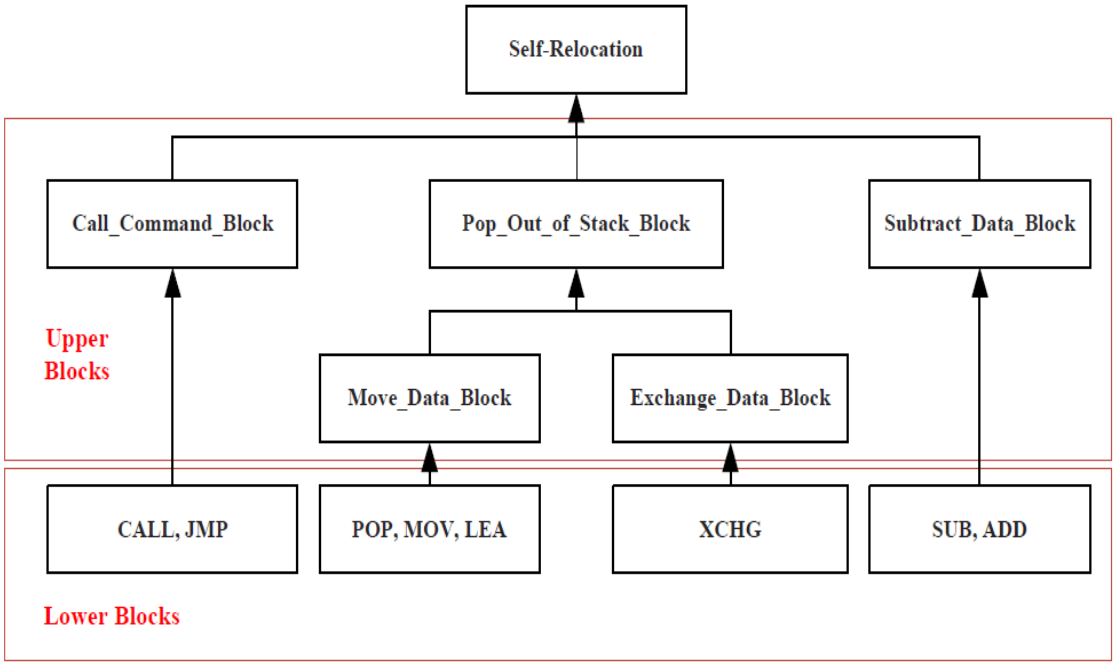

Figura 1.7. Estructura del SRG

Máquina detectora del SRG

Ahora podemos definir una máquina para detectar el SRG: **M = {Q, Σ, δ, q_0, F}** siendo,

Q conjunto de estados no vacíos.

Σ alfabeto de entrada.

δ función de transición (δ:QxΣ → Q)

q_0 estado inicial

F conjunto de estados finales.

Teniendo en cuenta la anterior representación del SRG nuestra máquina detectora quedaría de la siguiente manera:

$$M = \{V_T, V_N \cup \{T\}, \delta, S, F\}$$

Algoritmo detector del SRG

Para detectar el SRG podemos implementar un algoritmo que siga el siguiente diagrama de flujo:

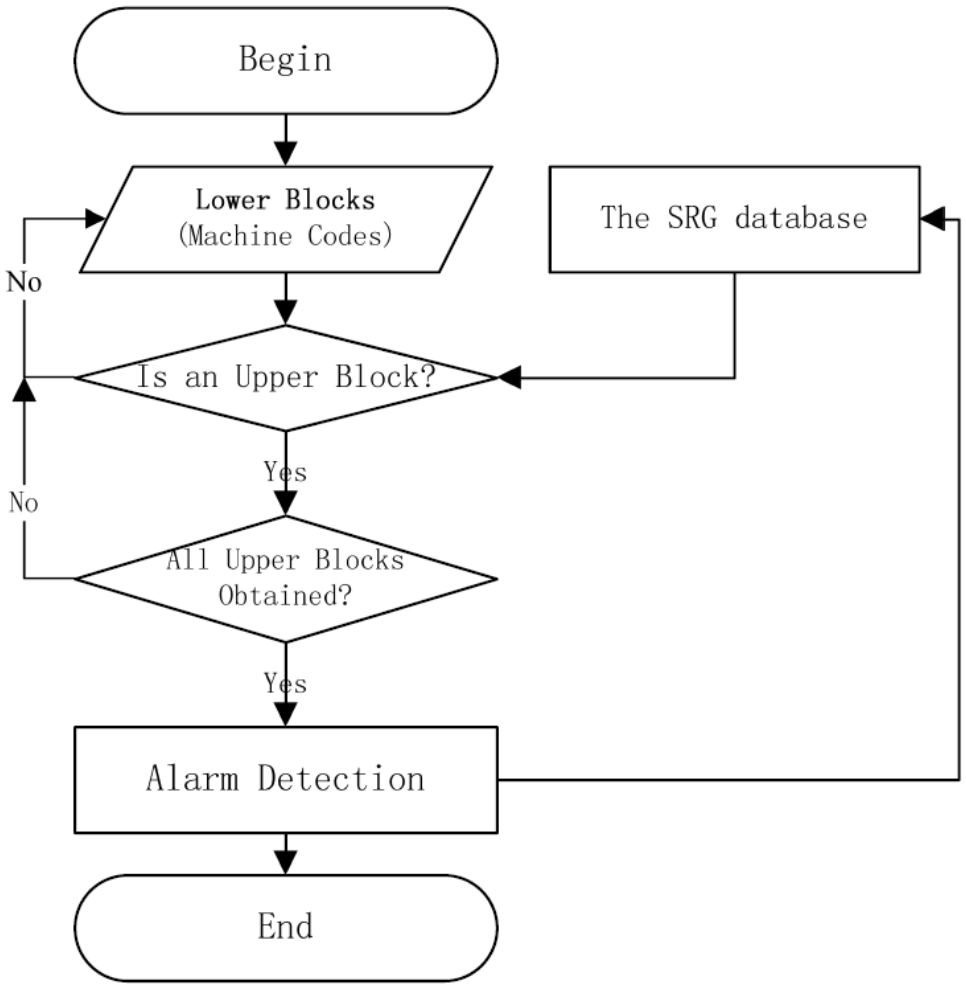

Figura 1.8. Algoritmo de detección del SRG

En primer lugar se divide el código en *Lower Blocks*. A continuación, se comparan estos bloques con la base de datos de SGR para determinar si puede formarse con ellos algún *Upper Block.* Más tarde, se comprueba si se han obtenido todos los *Upper Blocks* y si es así se genera una alarma. Por último, se guarda la información en la base de datos lo que permite al sistema aprender de manera autónoma.

Pruebas y resultados

Para realizar las pruebas se usó una colección de 100 virus extraídos de la web *VX Heavens* y 500 ejecutables (código no malicioso) tomados del *system32* de *Windows*. Los resultados obtenidos pueden verse en la siguiente gráfica:

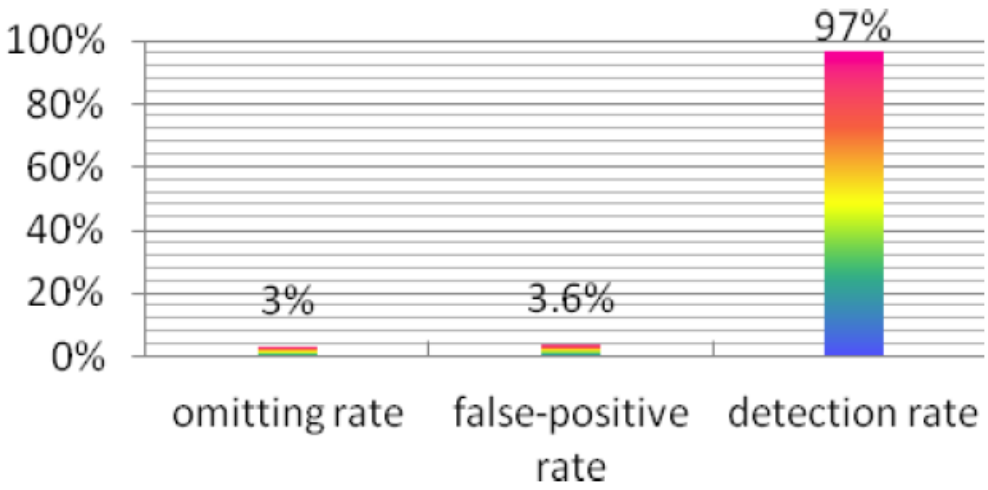

Figura 1.9. Gráfica de resultados del sistema

Los resultados son muy positivos. El 97% de los virus han sido detectados y solo se obtiene un 3% de falsas alarmas, un ratio bajo para ser un programa dirigido a la detección de los nuevos virus.

En el siguiente gráfico se compara este sistema con otros programas conocidos:

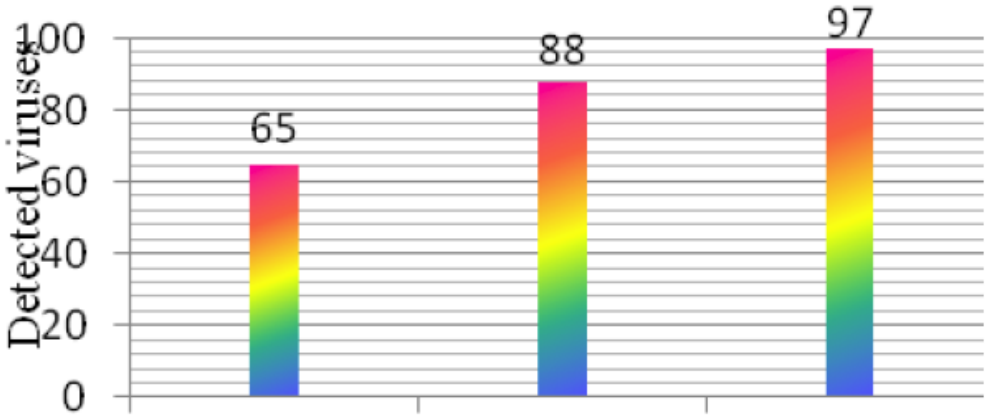

Figura 1.10. Comparación del sistema SRG con otros sistemas conocidos

4.3. Random Forest

Es un sistema híbrido que combina detección por patrones con detección por anomalías. Aunque también detecta código malicioso está pensado para detectar cualquier ataque o intrusión que pueda sufrir una red.

Este sistema se caracteriza por la utilización de *random forest* para la clasificación del tráfico durante la fase de entrenamiento y la fase de detección.

Descripción del algoritmo de clasificación

El algoritmo de clasificación del tráfico se basa en la creación de múltiples árboles de decisión.

Se trata de un algoritmo de clasificación basado en la creación de múltiples árboles de decisión. Estos árboles tienen características discriminantes en cada uno de sus nodos de forma que una muestra recorrerá el árbol desde la raíz hasta alguna de las hojas. En cada nodo intermedio, la muestra se evalúa y elige el camino que mejor se ajuste a sus características. Las hojas del árbol son las diferentes clases en las que se dividirá el tráfico, es decir que cuando una muestra llega a una hoja, ésta define la clase a la que pertenece.

Las ventajas de usar árboles de decisión para la clasificación es la rapidez en que se procesa el tráfico. Sin embargo, para mantener esta rapidez y no saturar el sistema los árboles no deben ser muy grandes ni complejos, lo que ocasiona poca precisión a la hora de clasificar. Para solucionar este problema se usa el algoritmo de *random forest.*

El algoritmo de *random forest* consiste en crear varios árboles de decisión distintos pero similares en cuanto a las características usadas en sus nodos. Para clasificar una muestra, se introduce en cada uno de estos árboles y una vez obtenido los resultados se calcula la moda, es decir, se calcula la hoja más repetida que ha alcanzado la muestra en los árboles. Esta hoja será la clase a la que pertenezca la muestra.

El algoritmo se caracteriza por dos parámetros fundamentales: el número de árboles que compondrán el bosque y las características que se usarán en los nodos para discriminar las muestras.

Además, para aumentar el rendimiento y la precisión del sistema se usan otras técnicas que se explican a continuación:

- El **mecanismo outlier** que se encarga de medir el grado de pertenencia de una muestra a una clase determinada. Se basa en el cálculo de un tipo de distancia entre la muestra un todas las clases. Se usa en el módulo se detección por anomalías.

- El algoritmo de **selección de características** se encarga de determinar cuáles serán las características que usen los árboles para clasificar los paquetes. Para ello analiza todas las características disponibles, las evalúa otorgándolas una puntuación y elige las más apropiadas, es decir las de mejor puntuación (normalmente se define un umbral para determinar si una característica es buena).

- Las **técnicas de sampling** se encargan de equilibrar las muestras de ataques usadas en el aprendizaje para obtener una mayor precisión en la detección posterior. Se intenta que la frecuencia de aparición sea parecida para entrenar al sistema contra los ataques poco comunes que suelen ser los más peligrosos.

4.3.1. Modelo de detección por patrones

El funcionamiento de este sistema tiene dos fases: offline y online.

La fase offline es la encarga de la creación de los patrones de los ataques conocidos que queremos detectar. El proceso de entrenamiento es supervisado ya que se usa tráfico con ataques y se debe indicar al sistema el tipo de ataque que se está clasificando en cada momento.

Antes de crear los patrones mediante el algoritmo de *random forest* se tienen que configurar los parámetros del sistema:

En primer lugar se usa el algoritmo de selección para escoger las características de comparación más apropiadas para este tipo de tráfico. Después se realizan varias pruebas cambiando el número de características seleccionadas y el número de árboles usados. Los parámetros que tengan mayor precisión con el tráfico de entrenamiento serán los utilizados.

Una vez hecho esto se utiliza el método de sampling para equilibrar la proporción de los distintos ataques. Es decir, se trata el tráfico de entrenamiento para que haya un número similar de ataques comunes (DOS, escaneo de puertos,…) y de ataques menos frecuentes (U2R, R2L,…).

Una vez asignados los parámetros y tratado el tráfico de entrenamiento se usa el algoritmo de random forest sobre la muestra para construir los patrones de los ataques. Estos patrones serán la entrada del módulo de detección en la fase online.

La fase online es la fase de detección. En ella se captura el tráfico y se preprocesa para obtener las características de los paquetes. Después, el tráfico pasa al módulo de detección que usa los árboles de decisión para clasificar el paquete y compararlo con los patrones para determinar si un paquete es tráfico legítimo o malicioso.

Experimentos y resultados

Para realizar las pruebas se usó el dataset KDD'99 (Knowledge, Discovery and Data-mining). Además de probar el detector se realizaron experimentos para demostrar el beneficio de preprocesar los datos con las técnicas explicadas anteriormente.

1) Comparación de rendimiento entre trafico equilibrado y desequilibrado

El test utilizado (10% set de entrenamiento) está desequilibrado (391.458 conexiones de DOS frente a 52 de U2R). Para equilibrarlo se selecciona aleatoriamente el 10% de las conexiones mayoritarias y se aumenta el número de las minoritarias replicándolas. El set final tiene 60.620 conexiones (es mucho más pequeño que el original).

El algoritmo se ejecuta usando parámetros básicos en los dos conjuntos de datos. Se comprueba como el resultado es mucho mejor cuando el tipo de ataque está equilibrado, sobre todo a la hora de detectar los ataques poco frecuentes.

2) Selección de características importantes

El dataset ofrece 41 características para la clasificación de los paquetes. El algoritmo de selección puntuará todas estas características y devuelve una lista ordenada. En las pruebas realizadas se observa se desecharon 3 características que obtenían mucha menor puntuación que el resto.

3) Optimización de parámetros

También se realizaron pruebas para determinar el valor de los parámetros que da mejor resultado. Se construyeron árboles con 5, 10, 15, 20, 25, 30, 35 y 38 de las características más importantes seleccionadas previamente. Después se analiza el tráfico de entrenamiento y se observa el rendimiento de cada tipo de árbol, es decir la tasa de detección y el tiempo empleado.

En las pruebas se observa que la tase de detección más alta se obtiene con 15, 25 y 30 características. Sin embargo, cuantas menos características haya menor será el tiempo empleado por lo que elegiríamos 15 como número de características.

4) Evaluación y discusión

Una vez que se han probado y elegido los mejores parámetros (en este caso 15 características y 50 árboles de decisión) se utiliza el dataset de KDD’99 para probar el sistema.

Los resultados obtenidos son muy buenos ya que se supera el mejor resultado obtenido hasta la fecha con ese dataset tanto ajustando los parámetros como ejecutándolo sin más.

4.3.2. Modelo de detección por anomalías

El funcionamiento es parecido al modelo de detección por patrones ya que también crea patrones y utiliza el algoritmo *random forest* para clasificar el tráfico.

En primer lugar, al igual que en el módulo anterior, se deben escoger los parámetros óptimos mediante el algoritmo de selección y las diferentes pruebas sobre el tráfico de entrenamiento.

La fase de entrenamiento consiste en crear patrones de tráfico libre de ataques para cada uno de los servicios que tiene la red: http, ftp, Telnet, pop, smtp,… Los patrones se construirá con el algoritmo *random forest* de la misma forma que se ha explicado para el módulo anterior. En este caso, el entrenamiento es no supervisado ya que el etiquetado del servicio al que pertenece cada paquete se puede hacer automáticamente mirando en la cabecera del paquete.

En la fase de detección se clasifica el tráfico y se compara con el patrón del servicio correspondiente. Si los datos obtenidos difieren mucho del comportamiento normal de ese servicio o encajan con el patrón de otro servicio se disparará una alarma indicando que puede existir un ataque. Para decidir que una conexión difiere mucho de su patrón de comportamiento se usa el mecanismo de outlier que calculará la distancia de la muestra con cada uno de los servicios existentes.

Experimentos y resultados

Para realizar las pruebas se escogieron los 5 servicios más populares en el uso de una red: ftp, http, pop, smtp y Telnet. Se creó un dataset con conexiones normales (sin ataques) de estos servicios. A partir de este dataset se generaron otros cuatro llamados 1%, 2%, 5% y 10% que poseían respectivamente un 1%, 2%, 5% y 10% de ataques en las conexiones.

Se usaron los 4 datasets con ataques para ver cómo respondía el sistema. Las pruebas mostraron que el rendimiento decrece a medida que aumenta la cantidad de ataques. Con un 1% de ataques la tasa de detección es del 95%, sin embargo con el dataset 10% la tasa es de 85%.

En cuanto a las intrusiones minoritarias, que son más difícil de detectar, el sistema tuvo un rendimiento mucho menos ya que la tasa de detección es solo del 65% con una tasa de falsos positivos del 1%. A pesar de ser un rendimiento menor es aceptable para este tipo de ataques.

4.3.3. Modelo híbrido

El modelo híbrido combina los dos modelos vistos anteriormente para detectar todo tipo de ataques ya sean conocidos o nuevos.

El tráfico pasará primero por el detector por patrones. Ahí se eliminarán los ataques detectados y el tráfico pasará al módulo por anomalías. Una vez ahí se comprobará si el tráfico es normal y en caso de que no lo sea y se detecte un ataque se enviará una alarma y se actualizará el patrón del módulo por patrones para incluir a este nuevo ataque.

El principal problema de este sistema es que el entrenamiento del módulo por anomalías se hace con la salida módulo por patrones que puede no detectar todos los ataques y permitir que el sistema se entrene con tráfico malicioso.

En la figura 1.11. se muestra un diagrama de flujo que explica el comportamiento de este sistema híbrido.

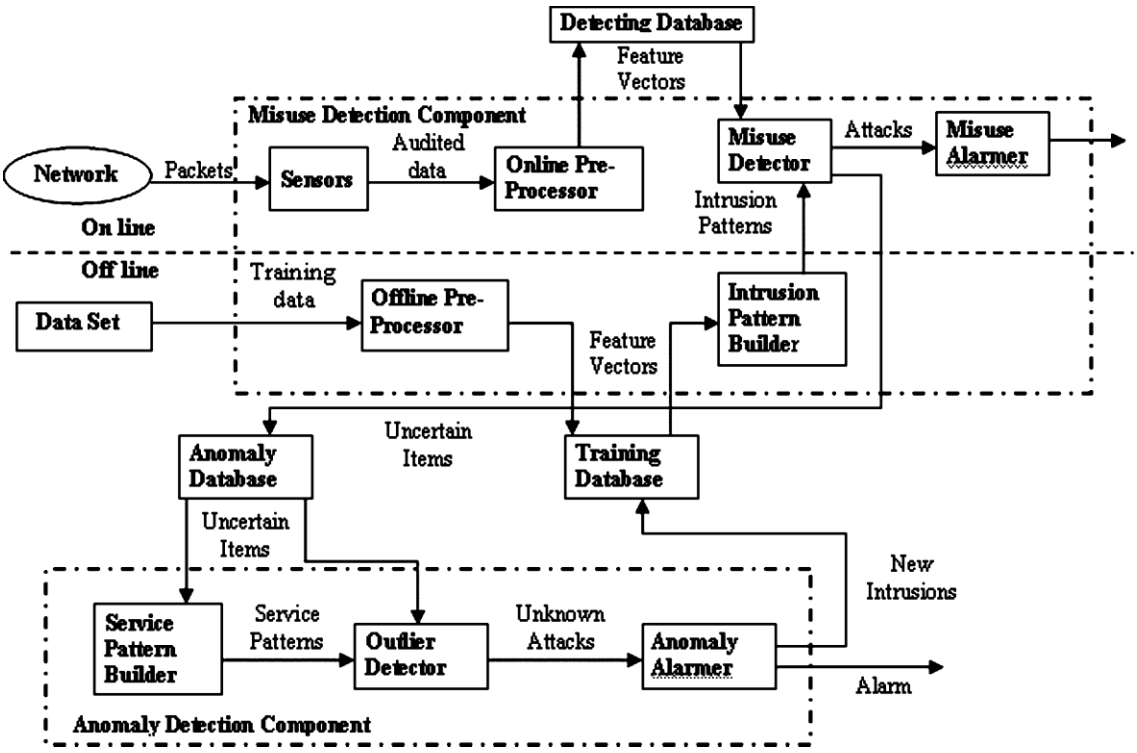

Figura 1.11. Funcionamiento del sistema híbrido random forest

4.4. Sistemas inmunológicos artificiales

Una de las técnicas investigadas en los últimos años consiste en imitar el sistema biológico inmunitario (SBI), es decir en tratar una intrusión en un sistema informático de la misma forma que los seres vivos tratan sus infecciones.

El SBI es capaz de detectar organismos extraños y eliminarlos sintetizando unas proteínas llamadas anticuerpos que tienen, a su vez, varios tipos de detectores. Cuando un anticuerpo detecta un intruso, se multiplica para ser más efectivo hasta que la amenaza es eliminada. Después de esto la mayoría de los anticuerpos mueren aunque algunos permanecen, creando una memoria inmunitaria para poder reaccionar rápidamente en caso de contraer de nuevo la infección.

Las técnicas que se muestran a continuación intentar imitar este comportamiento para detectar el código malicioso.

4.4.1. Sistema Inmune Adaptativo (AIS)

Este sistema usa redes neuronales para crear los detectores. Dispone de dos fases en su funcionamiento: una de aprendizaje y otra de detección.

Algoritmo de aprendizaje

La red neuronal usada en este sistema tiene 3 capas: capa de entrada, una capa competitiva oculta (capa de Kohonen) y una capa de salida.

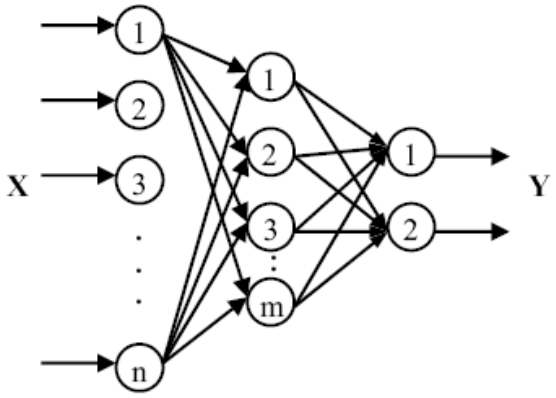

Figura 1.12. Red neuronal usada por el sistema AIS

La capa de entrada tiene tantos nodos como elementos tiene cada muestra. Por ejemplo, si queremos comparar bit a bit y cada muestra de código que tomamos es de 128 bits la capa de entrada tendrá 128 elementos.

La capa intermedia es la capa de procesamiento. Sus nodos tendrán pesos a la entrada (salida de la capa de entrada) que se irán modificando en el aprendizaje para clasificar las muestras en clases.

La capa de salida tiene tantos nodos como número de clases finales, en nuestro caso dos: código malicioso y código seguro. Las clases de salida forman los *code vectors*. Estas clases vendrán definidas por un centro que las agrupa de forma que este centro variará con cada nueva muestra procesada.

Como método de aprendizaje se usa el método de entrenamiento competitivo con un ganador que tiene las siguientes fases (n = número de nodos en la capa inicial, m = número de nodos en la capa intermedia):

1. Se generan aleatoriamente pesos para la capa intermedia con valores en el rango [0,1]

2. Se define un valor inicial para el tiempo t = 1

3. Para todas las muestras de entrada x_i se calcula:

   a. La norma del vector $D_j^i = \left| x_i - w_j \right|$ donde $j \in 1..m$

   b. La neurona o elemento de la capa intermedia que gana es aquel k que cumple: $D_k^i = \min_j D_j^i$

   c. Se modifican los pesos de la red neuronal de la siguiente forma: $w_{ij}(t+1) = w_{ij}(t) + \gamma(t) \cdot (x_i - w_{ij}(t))$ si j = k $w_{ij}(t+1) = w_{ij}(t)$ si $j \neq k$ donde $i \in 1..n$ $j \in 1..m$. El valor de γ es lo que caracteriza el aprendizaje y puede o ser constante o decrecer con el tiempo según la función $1/t$.

4. Se cambia el valor de t por t+1 y se repite el proceso desde el paso 3

Proceso de detección

El proceso de detección tiene 5 fases:

1. Generación de los detectores

   Los detectores serán vectores de peso LVQ (cadenas de bits) creados aleatoriamente.

2. Maduración de los detectores

   Para entrenar al sistema se seleccionan cadenas de bits tanto de programas seguros como de código malicioso (usar varios tipos para hacer más eficiente el entrenamiento). A continuación, se ejecuta el algoritmo de aprendizaje para modificar los pesos de la red neuronal y hacer evolucionar los detectores. Como cada detector

está entrenado con diferentes muestras será más efectivo para detectar un tipo determinado de ataques.

3. Proceso de detección

   En esta fase el tráfico pasará a través de los detectores que lo clasificarán como código legítimo o código malicioso. Además, los detectores ineficientes se eliminan. Cada detector posee un ciclo de vida después del cual desaparece dejando paso a otros más recientes y mejor preparados.

4. Clonación y mutación de detectores

   Cuando se detecta algún código malicioso, debido a que este tiende a expandirse por otros archivos, se clona al detector que lo ha encontrado para acelerar la eliminación de la amenaza. Además, durante estas clonaciones puede haber pequeñas mutaciones para buscar otros detectores más efectivos.

5. Creación de una memoria inmunitaria

   Después de que el sistema haya localizado y eliminado todo el código malicioso, los mejores detectores de este tipo son guardados en una memoria inmunitaria. De esta forma, se podrán utilizar más adelante ataques del mismo tipo.

En la figura 1.13. puede verse gráficamente el proceso de detección.

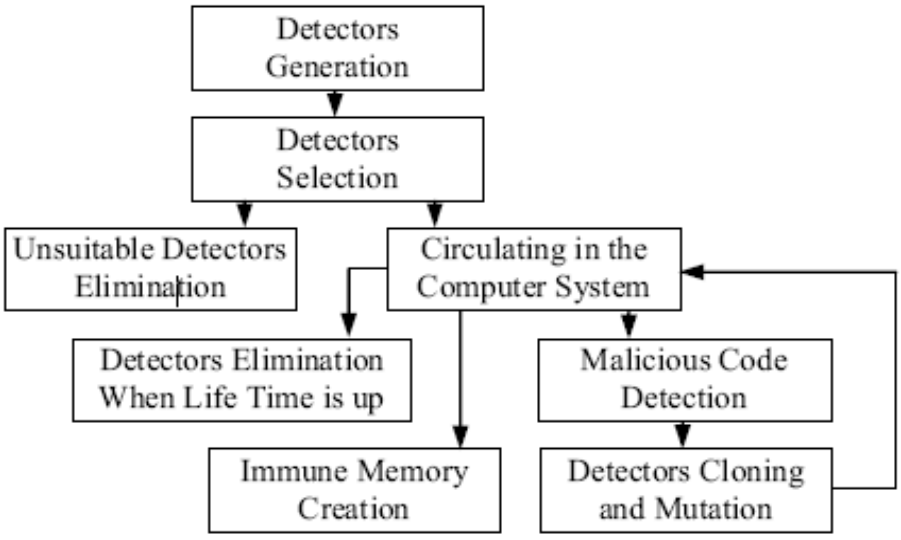

Figura 1.13. Proceso de detección del sistema AIS

Implementación estudiada

La implementación estudiada de este modelo es la siguiente.

La estructura LVQ tiene 128 elementos (neuronas) en la capa de entrada, 10 elementos en la capa intermedia y 2 en la de salida.

Las muestras de aprendizaje para cada detector son 4 archivos propios seguros y uno con código malicioso seleccionados aleatoriamente. De cada archivo se seleccionan 5 fragmentos de 128 bits que se usarán para el aprendizaje.

Durante el aprendizaje se dice que fragmentos son buenos y cuáles no. Como resultado obtenemos, en la capa media de la red neuronal, 8 nodos que se corresponden con la clase de código seguro y 2 con la de código malicioso.

En la fase de detección cada archivo se fragmenta en trozos de 128 bits y se obtiene la clasificación de cada uno de estos fragmentos. A continuación se observa el porcentaje de cada clase. Si el porcentaje de fragmentos legítimos es mayor que 80% el archivo se considera válido, si es menor el archivo se considera código malicioso.

Por ejemplo el archivo diskopy.com se descompone en 56 fragmentos. El detector clasifica 49 de ellos como seguros (87.5%) y 7 como maliciosos (12.5%).Como el número de fragmentos seguros es superior al 80% este archivo se considera seguro.

Resultados Experimentales

Para el desarrollo del algoritmo se utilizó Matlab v6.5, que implementa una librería de redes neuronales.

Para la realización de las pruebas se escogieron utilidades del sistema operativo y algunos virus conocidos. Los resultados se muestran en las siguientes tablas (Ps es la probabilidad de que el archivo sea seguro y Pm de que sea malicioso).

G

File name	Detector 1 P_S / P_M	Detector 2 P_S / P_M	Detector 3 P_S / P_M	Detector 4 P_S / P_M
cacls.exe	0,78 / **0,22**	**0,93** / 0,07	**0,89** / 0,11	**0,82** / 0,18
ctfmon.exe	**0,81** / 0,19	**0,86** / 0,14	**0,87** / 0,13	**0,89** / 0,11
dbexplor.exe	**0,90** / 0,10	**0,93** / 0,07	**0,94** / 0,06	**0,90** / 0,10
dcomcnfg.exe	**0,91** / 0,09	**0,96** / 0,04	**0,96** / 0,04	**0,96** / 0,04
diskcopy.com	**0,83** / 0,17	**0,93** / 0,07	**0,92** / 0,08	**0,83** / 0,17
dllhost.exe	**0,89** / 0,11	**0,96** / 0,04	**0,98** / 0,02	**0,85** / 0,15
etm70.exe	**0,91** / 0,09	**0,94** / 0,06	**0,95** / 0,05	**0,87** / 0,13
notepad.exe	**0,84** / 0,16	**0,91** / 0,09	**0,92** / 0,08	**0,83** / 0,17
soundman.exe	**0,87** / 0,13	**0,93** / 0,07	**0,94** / 0,06	**0,93** / 0,07
taskman.exe	**0,88** / 0,12	**0,92** / 0,08	**0,95** / 0,05	**0,92** / 0,08
uninlib.exe	0,58 / **0,43**	**0,81** / 0,19	**0,83** / 0,17	**0,82** / 0,18

Tabla 1.5. Resultados de las pruebas con archivos no infectados

File name	Detector 1 P_S / P_M	Detector 2 P_S / P_M	Detector 3 P_S / P_M	Detector 4 P_S / P_M
Backdoor.Agent	**0,98** / 0,02	**0,98** / 0,02	**0,98** / 0,02	**0,96** / 0,04
Backdoor.Agobot	**0,91** / 0,09	0,58 / **0,42**	0,68 / **0,32**	**0,83** / 0,17
E-Worm.Bozori	0,64 / **0,36**	0,73 / **0,27**	0,55 / **0,45**	**0,85** / 0,15
E-Worm.Zafi	0,70 / **0,30**	0,58 / **0,42**	0,68 / **0,32**	**0,87** / 0,13
E-Worm.Mydoom	0,67 / **0,13**	0,65 / **0,35**	0,65 / **0,35**	0,79 / **0,21**
E-Worm.NetSky	0,61 / **0,39**	0,68 / **0,32**	0,57 / **0,43**	0,80 / 0,20
Exploit.DebPloit	**0,85** / 0,15	**0,92** / 0,08	**0,92** / 0,08	**0,92** / 0,08
N-Worm.Lovesan	**0,83** / 0,17	**0,81** / 0,19	0,77 / **0,23**	0,71 / **0,29**
Net-Worm.Mytob	**0,84** / 0,16	0,55 / **0,45**	0,63 / **0,37**	0,74 / **0,26**
Trojan.Bagle	**0,81** / 0,19	**0,85** / 0,15	0,78 / **0,22**	0,68 / **0,32**
Trojan.Daemoniz	**0,93** / 0,07	**0,84** / 0,16	**0,84** / 0,16	**0,84** / 0,16
Trojan.LdPinch	**0,89** / 0,11	0,60 / **0,40**	0,76 / **0,24**	**0,81** / 0,19
Virus.Gpcode	0,73 / **0,27**	0,54 / **0,46**	0,64 / **0,36**	0,58 / **0,42**
Virus.Hidrag	0,79 / **0,21**	0,76 / **0,24**	0,75 / **0,25**	0,77 / **0,23**

Tabla 1.6. Resultados de las pruebas con código malicioso

El primer detector se entrenó con *E-Worm.Win32.Mydoom* como código malicioso. Puede verse como detecta con éxito los E-Worms y los Virus (Gpcode, Hidrag). Sin embargo da falsos positivos con *uninlib.exe* y *cacls.exe* de forma que sería eliminado en la fase de selección.

En la tabla 1.6. se puede observar como cada detector encuentra tipos de ataques diferentes. Esto se debe al entrenamiento que haya tenido cada uno. Con un número mayor de detectores se conseguiría un resultado mucho mejor.

En la siguiente tabla se muestran una comparación de resultados con otros métodos de detección. Se observa que el mejor es el antivirus completamente actualizado pero el rendimiento de AIS es muy bueno teniendo en cuenta que no tiene ninguna información sobre los ataques.

File name	Kaspersky antivirus (actual bases)	Kaspersky antivirus (outdated bases)	NOD32 (heuristic analyzer)	AIS (four detec-tors)
Backdoor.Agent.lw	Backdoor	OK	OK	OK
Backdoor.Agobot	Backdoor	Backdoor	Agobot	Virus
Email-Worm.Maddas	Email-Worm	Email-Worm	OK	Virus
Email-Worm.Gigger	Email-Worm	Email-Worm	OK	Virus
Email-Worm.Loding	Email-Worm	Email-Worm	OK	Virus
Email-Worm.Zafi.d	Email-Worm	OK	Zafi	Virus
Net-Worm.Bozori.a	Net-Worm	OK	Bozori	Virus
Net-Worm.Mytob.a	Net-Worm	OK	Mytob	Virus
Trojan.Psyme.y	Trojan	OK	OK	Virus
Trojan.Bagle	Trojan	OK	Bagle	Virus
Trojan.Agent	Trojan	Trojan	OK	Virus
Trojan.Daemonize	Trojan	Trojan	OK	OK
Trojan.Mitglieder	Trojan	Trojan	Trojan	Virus
Trojan.LdPinch	Trojan	Trojan	PSW	Virus
Virus.Gpcode.ac	Virus.Win32	OK	OK	Virus
Exploit.DebPloit	Exploit	OK	OK	OK

Tabla 1.7. Comparativa entre el sistema AIS y otros antivirus

4.4.2. Sistema basado en perfiles estadísticos

Inspirado en el sistema inmunológico humano, este sistema, llamado MaCDI, pretende detectar malware aunque este sea desconocido usando un perfil de llamadas al sistema.

Los desafíos de esta arquitectura son:

- La capacidad de aprender y anticipar el nuevo comportamiento del malware

- Minimizar el impacto del rendimiento filtrando las llamadas al sistema más usuales para reducir el tamaño de los perfiles

- Minimizar el impacto del rendimiento al hacer las copias de seguridad de los archivos

Esta propuesta utiliza técnicas basadas en perfiles estadísticos en vez de usar inteligencia artificial. En lugar de buscar patrones en el código malicioso, este sistema busca patrones en las llamadas al sistema que hacen los programas.

Arquitectura

La arquitectura de este tipo de sistemas se puede ver en la figura 1.14.

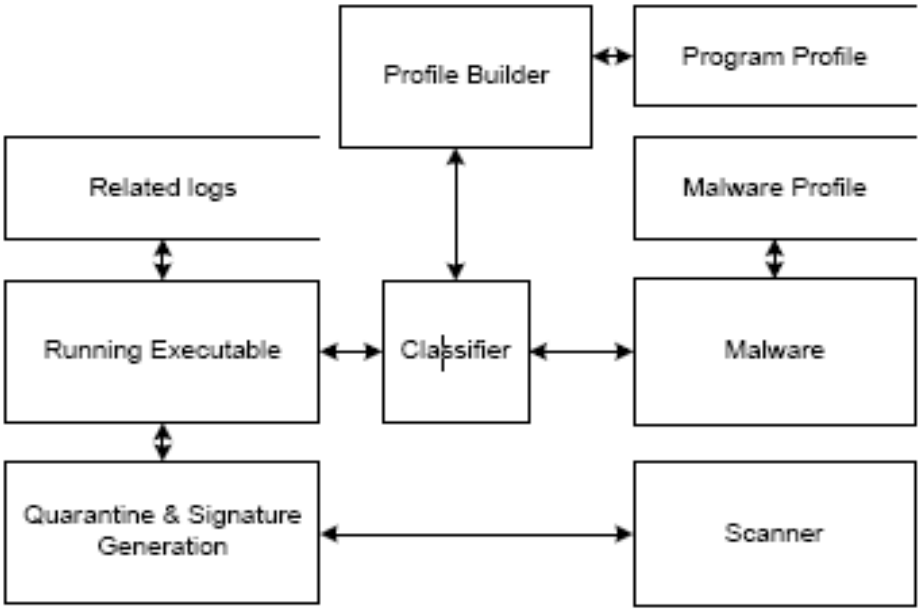

Figura 1.14. Componentes del sistema MaCDi

En primer lugar, se crean los perfiles con las llamadas al sistema que usan los programas de código malicioso. Para mejorar la eficiencia de estos perfiles se eliminarán las llamadas que realizan los programas legítimos.

El algoritmo tiene dos fases: la fase de adolescencia y la fase de madurez. Estas fases se corresponden con las fases innata y adquirida del sistema inmunológico humano.

Fase de Adolescencia

Cada nuevo programa instalado en el sistema entra en la fase de adolescencia. Una vez aquí todas las llamadas al sistema que haga se registrarán y compararán con el perfil de malware creado en la fase de entrenamiento. Si estas llamadas se corresponden con algún patrón sospechoso se abortará la ejecución del programa y el código pasará a estar en cuarentena.

A continuación, se crea un resumen del código de ese programa y se buscan programas similares en los directorios relacionados. Si se encuentra alguno también es puesto en cuarentena.

Si no se encuentra ninguna llamada sospechosa se crea un perfil con las llamadas realizadas durante un determinado periodo de tiempo. El programa pasará a la fase de madurez cuando se cumplan ciertos requisitos como la cantidad de código ejecutado o la creación de un perfil lo suficientemente exhaustivo.

En la figura 1.15 puede verse el diagrama de funcionamiento del sistema en la fase de adolescencia.

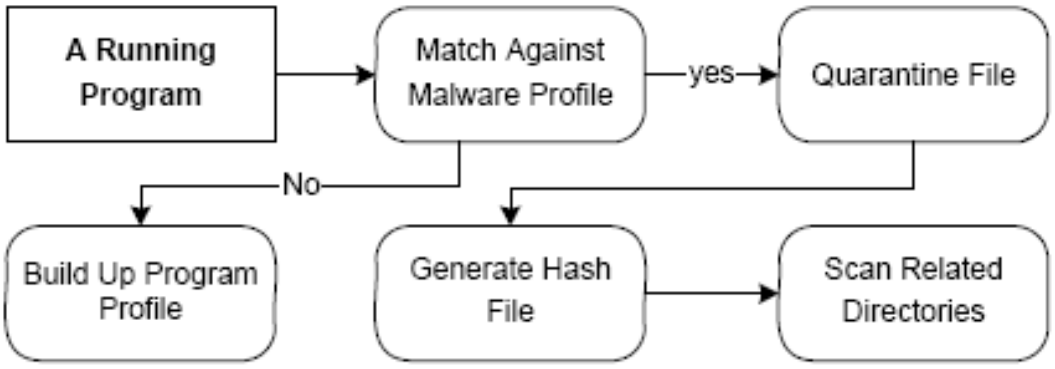

Figura 1.15. Diagrama fase de adolescencia del sistema MaCDi

Fase de madurez

Durante la fase de madurez, el comportamiento del programa se compara con el perfil realizado durante la fase de adolescencia. Si las llamadas realizadas con las esperadas el programa sigue su ejecución. Sin embargo, si se detecta un comportamiento anómalo el programa pasará de nuevo a la fase de adolescencia donde se evaluará su ejecución.

El diagrama de funcionamiento de esta fase es el siguiente:

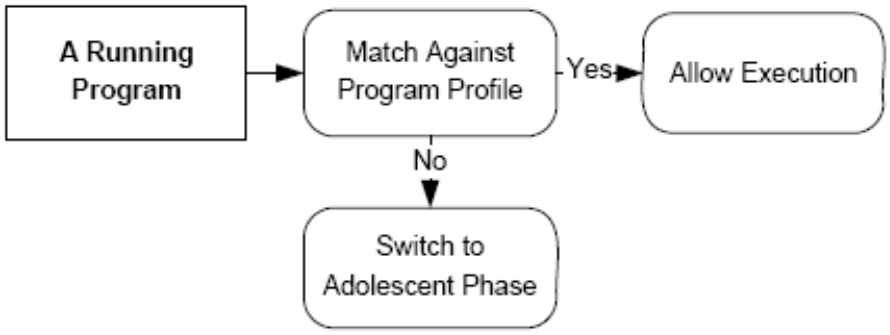

Figura 1.16. Diagrama fase de madurez del sistema MacDi

Todas las operaciones que impliquen una modificación de un archivo guardarán una copia de seguridad del estado anterior. Si durante el análisis se comprueba que la ejecución del programa es legítima estas copias de seguridad se borran. Sin embargo, si se detecta código malicioso los archivos modificados por el programa se restaurarán a su estado anterior.

Para la implementación de este sistema se recomienda usar un sistema de archivos con copy-on-write, ofrecido por la mayoría de sistemas operativos. Este sistema guarda los archivos eliminados durante un periodo de tiempo por si necesitan ser restaurados.

La copia de todos los archivos modificados y eliminados puede suponer un gran coste computacional. Para mejorar el rendimiento los archivos se clasifican en 3 niveles de prioridad:

- Archivos de Usuario (Prioridad Alta)
- Archivos del Sistema Operativo (Prioridad Media)
- Archivos de Aplicaciones (Prioridad Baja)

Si el rendimiento del sistema baja, los archivos menos prioritarios dejarán de copiarse para disminuir el coste computacional.

4.5. Otras propuestas

- **Sistema con archivos de firmas basados en n-gram**

Esta propuesta solo puede usarse como HIDS ya que requiere un gran coste computacional y no podría analizar todo el tráfico que pasa por una red.

La fase de entrenamiento consiste en guardar en archivos todos los n-gram de diferentes programas legítimos y maliciosos. Para cada programa se tendrá un archivo lo que conlleva la necesidad de una memoria muy grande. Además el sistema usa n-grams de tamaño 6, e incluso 8, lo que aumenta en gran medida el coste computacional.

Durante la fase de detección el código que queremos analizar se compara con los archivos creados en la fase de entrenamiento mediante el uso del algoritmo k-NN (*k nearest neighbours*). Este algoritmo devuelve los k archivos más parecidos al código analizado comparando todos sus n-grams. Una vez elegidos los k archivos, se mira su naturaleza y se decide si el programa analizado es legítimo o puede ser peligroso para el sistema.

- **Detección mediante estrategia de planificación de recursos**

Proponen dividir el tráfico según el protocolo al que pertenece (ICMP, UDP, TCP, etc.) para reducir la necesidad de procesamiento de los NIDS.

- **Modelo de integración de detección por firmas y prevención de anomalías**

Esta propuesta consiste en la creación de un NIDS híbrido que permita combinar en un solo programa las ventajas de los detectores basados en

firmas (buen rendimiento pero solo ante ataques ya conocidos) y de los basados en anomalías (efectivos ante nuevos ataques aunque con menos precisión).

Como sistema de detección basado en firmas se usará Snort. Además se utilizará la información de etiquetamiento del tráfico que hace Snort para la toma de decisiones del sistema.

Para analizar el tráfico en busca de anomalías se utilizan redes bayesianas, debido a su habilidad para extrapolar conocimiento y aplicarlo en casos no vistos previamente. Durante la fase de entrenamiento se definirán las diferentes clases en las que podrá clasificarse el tráfico. Durante la detección, el tráfico a analizar será clasificado por las redes bayesianas como legítimo o malicioso.

Es preciso determinar unos umbrales para lograr un equilibrio entre las tasas de falsos positivos y falsos negativos. Por lo general, estos umbrales se determinarán de forma empírica, es decir haciendo pruebas para ver cuáles dan mejor resultado, ya cada red tendrá un tipo de tráfico y los umbrales serán distintos.

- **Rate limiting**

Detecta comportamiento anómalo en las conexiones basándose en la premisa que un host infectado tratará de conectarse a muchas maquinas diferentes en un periodo corto de tiempo. Esta propuesta detecta PortScans colocando a aquellas conexiones que exceden cierto umbral en una cola.

- **Algoritmo Threshold Random Walk (TRW)**

Se basa en que la probabilidad de que un intento de conexión sea exitoso es mayor para un programa benigno que para uno anómalo. Utiliza pruebas de hipótesis secuenciales para clasificar si un host remoto es un scanner o no.

- **TRW con un ratio límite (TRW-CB)**

Es una solución hibrida que busca complementar las dos soluciones anteriores. Es un detector que limita la tasa en la que nuevas conexiones son inicializadas aplicando pruebas de hipótesis secuenciales en orden inverso al cronológico. El algoritmo frenara aquellos hosts que han tenido conexiones fallidas.

- **Método de máxima entropía**

Estima la distribución del tráfico benigno usando una estimación de máxima entropía. En el periodo de entrenamiento se clasifica a los paquetes en x clases distintas y se determina una distribución para cada clase. Luego se observa la distribución de los paquetes en tiempo real y se comparan con la distribución base. Se dispara una alarma cuando una clase supera un umbral de diferencia.

- **Detección de anomalías en la cabecera de los paquetes (PHAD)**

Utiliza algunos campos del encabezado (direcciones IP, puertos, protocolos y banderas TCP). Durante la etapa de entrenamiento se le asigna un score a cada valor de cada uno de los campos del encabezado y luego se suman para obtener un score total de anomalía para cada paquete. Los primeros n paquetes con score más alto son considerados peligrosos.

- **Sistema experto de detección de intrusiones de nueva generación (NIDES)**

Es un detector estadístico que compara los perfiles de tráfico recolectados por largo tiempo con perfiles recolectados en corto tiempo sobre datos en tiempo real. Reporta una alarma si existe una desviación considerable entre ambas distribuciones.

III

IDS BASADO EN REGLAS: SNORT

1. INTRODUCCION

Snort es un Sistema de Detección de Intrusiones basado en red (NIDS o IDS de red) de código abierto que puede descargarse desde la página web www.snort.org. Es capaz de realizar un análisis en tiempo real del tráfico de una red así como registrar los paquetes que conforman dicho tráfico.

Implementa un motor para la detección de ataques que permite registrar, alertar y responder ante cualquier anomalía previamente definida. Para ello utiliza patrones (reglas) que corresponden a ataques, barridos, intentos de aprovechar alguna vulnerabilidad, análisis de protocolo, etc. Además existen una serie de herramientas que pueden utilizarse con Snort, ya sean herramientas para simplificar el uso del mismo o herramientas que trabajan conjuntamente y lo hacen más potente.

Snort utiliza un lenguaje flexible de reglas para describir el tráfico e indicar qué paquetes pueden pasar y cuáles no (son considerados potencialmente peligrosos).

Además de realizar el análisis del tráfico en tiempo real, Snort tiene un módulo con capacidad de generar alertas en tiempo real, registrando dichas alertas en ficheros de texto ASCII, UNIX sockets, bases de datos….

Actualmente, Snort es uno de los NIDS basados en reglas más utilizado y eficaz por lo que es un gran programa de base a la hora de crear la protección de una red. Entre sus ventajas están:

- Es software libre.
- Es configurable. Snort se puede configurar para que se adapte a tu propia red, incluso puedes crear tus propias reglas (o realizar modificaciones de las existentes).
- Es muy popular.
- Se puede ejecutar en múltiples plataformas.
- Se actualiza constantemente.

- Existe una amplia documentación así como foros de debate de la aplicación.
- Es una herramienta con un amplio recorrido en el mundo de la seguridad informática, disponible libremente bajo licencia GPL desde 1998.

2. ARQUITECTURA DE SNORT

2.1. Arquitectura modelo de un IDS basado en reglas

Normalmente la arquitectura de un IDS basado en reglas está formada por los siguientes conjuntos:

1. La fuente de recogida de datos. Estas fuentes pueden ser un log, un dispositivo de red, o el propio sistema (los IDS basados en host).

2. Reglas que contienen los datos y patrones para detectar anomalías de seguridad en el sistema.

3. Filtros que comparan los datos recogidos con los patrones almacenados en las reglas.

4. Detectores de eventos anormales en el tráfico de red.

5. Dispositivo generador de informes y alarmas. En algunos casos son lo suficientemente sofisticados como para enviar alertas vía mail o SMS.

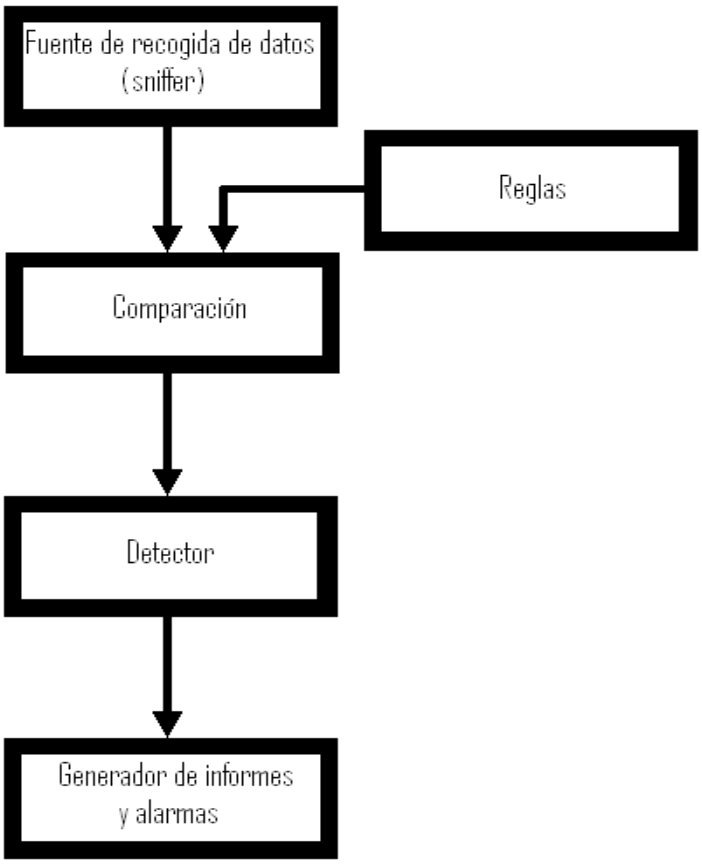

Figura 2.1. Arquitectura de un IDS basado en reglas

2.2. Arquitectura de Snort

La arquitectura presentada anteriormente ha de tomarse como ejemplo académico. Cada IDS implementa su arquitectura de manera diferente aunque en general todos toman como base el esquema anterior.

En el caso de Snort, su arquitectura se ajusta a este modelo pero con unas pequeñas diferencias:

- Después de la fuente de datos Snort incluye un **decodificador** de paquetes para facilitar su análisis posterior por el resto de los módulos.
- Para mejorar el análisis de los paquetes Snort incluye **preprocesadores** que pueden activarse o desactivarse según nuestras necesidades, mejorando los resultados finales obtenidos en la detección de ataques.
- En Snort los módulos de comparación y de detección se fusionan en un solo módulo, el **Detection Engine**.
- A su vez, el Detection Engine puede recibir soporte de otros detectores auxiliares para la detección de ataques específicos, los **Detection plug-in**.

La arquitectura de Snort está compuesta por los siguientes elementos:

- **Packet Capture Library**

Se encarga de la **captura** de paquetes. Para ello usa las librerías libpcap (en Linux) y WinPcap (en Windows).

- **Packet Decoder**

Toma los paquetes y los decodifica. Primero el Data Link frame, después el IP protocol y, por último los paquetes TCP o UDP. Cuando

termina la decodificación, Snort tiene toda la información de los protocolos en los lugares adecuados para un posterior análisis.

- **Preprocessor**

El preprocesamiento permite a Snort analizar posteriormente los paquetes más fácilmente. Puede generar alertas, clasificar los paquetes o filtrarlos.

- **Detection Engine**

Analiza los paquetes que le llegan de acuerdo a las reglas que tiene especificadas.

- **Detection (plug-ins)**

Módulos de detección auxiliares al Detection Engine para análisis más específicos.

- **Output**

Cuando salta una alarma Snort tiene diversos métodos de salida logins, databases and syslogs.

En la figura 2.2. se puede ver cómo están conectados todos estos módulos y, en definitiva, la arquitectura de Snort de una forma más gráfica.

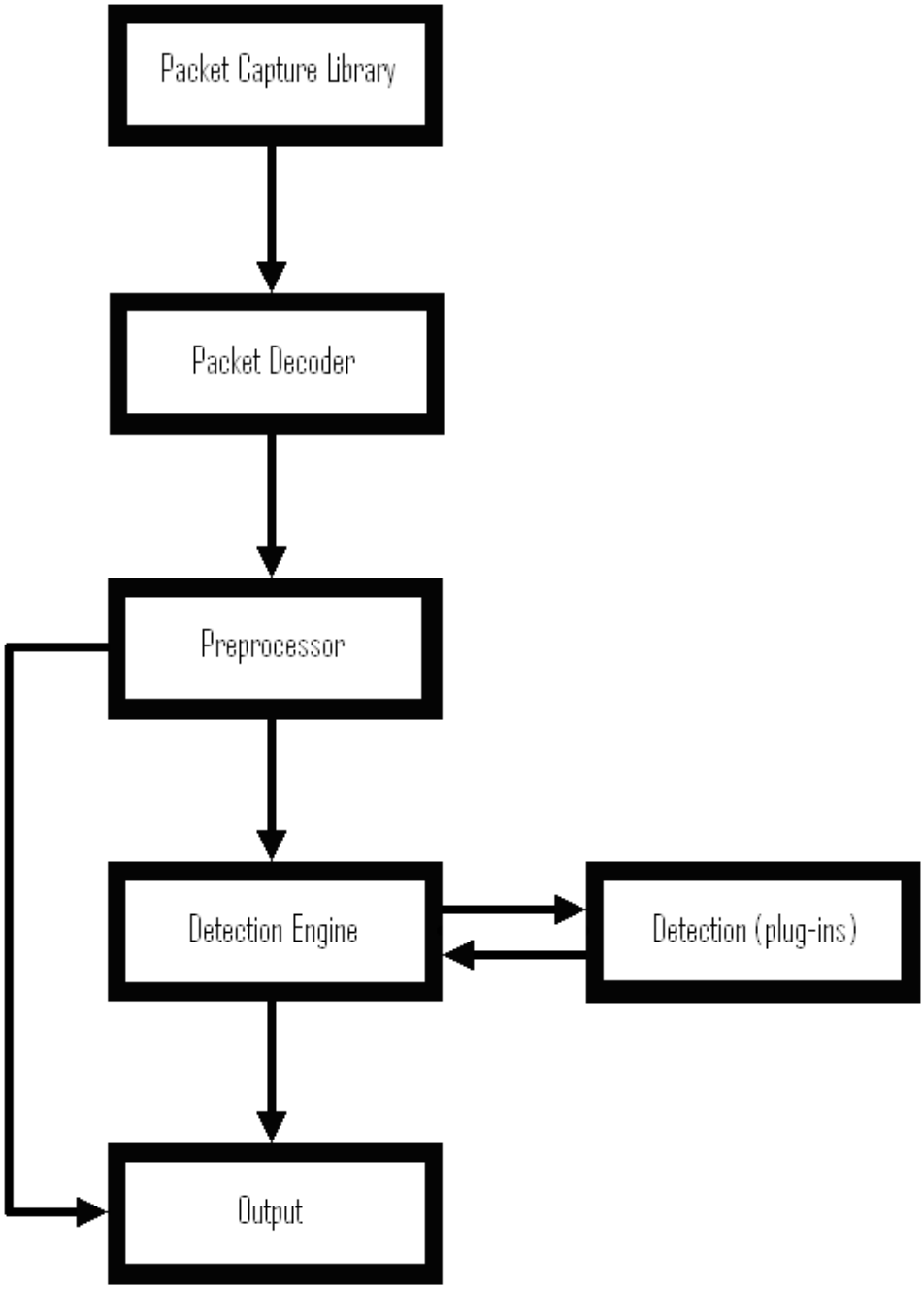

Figura 2.2. Arquitectura de Snort

2.3. Recorrido de los paquetes en Snort

En este apartado veremos el camino que recorren los datos dentro de la arquitectura de Snort:

1. Se capturan los paquetes provenientes de la red (Packet Capture Library) y se envían al Packet Decoder.

2. Los paquetes se decodifican basándose en el protocolo al que correspondan (Packet Decoder). Una vez decodificados, los paquetes son enviados a los Preprocessor´s.

3. Los paquetes son clasificados y filtrados. También es posible que ya se generen algunas alarmas (Preprocessor). Las alarmas generadas serán notificadas al módulo Output (y se le adjuntará el paquete que generó dicha alarma) y los paquetes ya clasificados y filtrados se envían al Detection Engine.

4. Se analizan los paquetes clasificados y filtrados en el Detection Engine. Dicho análisis se realiza mediante la búsqueda de correspondencias entre dichos paquetes y el juego de reglas con el que trabaja Snort (para saber más acerca de las reglas ver el apartado “Reglas”). A su vez el Detection Engine recibirá el soporte de los Detection plug-ins en caso de que sea necesario. Si este análisis hiciese saltar alguna alerta se notificaría al módulo output y se le adjuntaría el paquete que la generó. En caso contrario se permite al paquete el paso al sistema.

5. Una vez que llega el paquete que generó la alarma al modulo *Output*, este se encarga de actualizar el fichero *Alerts.ids* (en el que se almacena qué paquete generó una alerta y por qué) y el fichero *Snort.log.xxxxx* (en el que se almacena el contenido de dicho paquete de una forma más o menos detallada en función del formato de las alertas en el que se estuviese ejecutando Snort). Pasa saber más acerca del fichero *Alerts.ids* ver el apartado “Fichero de alertas generado por Snort”. Para saber más acerca del fichero *Snort.log.xxxxx* o del formato de las alertas ver el apartado “Formato de las alertas”.

3. PREPROCESADORES

Los preprocesadores fueron introducidos en la arquitectura de Snort en la versión 1.5 (Diciembre 1999). Extienden la funcionalidad de Snort permitiendo a los usuarios añadir plugins de manera sencilla.

El código de los preprocesadores se ejecuta antes del Detection Engine, pero después de que los paquetes hayan sido decodificados. Los paquetes pueden ser modificados o analizados de forma paralela usando este mecanismo.

Puede verse la configuración de los preprocesadores de Snort en el archivo *Snort.conf* en la sección 3 (*Step #3: Configure Preprocessors*).

A continuación mostramos la lista de los 15 preprocesadores con los que cuenta la versión 2.8 de Snort:

3.1. Frag3

El preprocesador frag3 es un módulo cuyo objetivo principal es la detección de ataques que intentan evadir a Snort mediante fragmentación. Reconstruye el flujo lógico de datos haciendo de esta manera más eficiente la detección de los ataques.

3.2. Stream5

Stream5 es un módulo de reensamblado TCP basado en objetivos. Es capaz de seguir la pista de la sesión tanto para el protocolo TCP como para los protocolos UDP e ICMP. Incluye acciones basadas en objetivos para tratar coincidencias de datos (overlapping data) y otras anomalías TCP.

Stream5 permite a otros protocolos normalizadores o preprocesadores configurar dinámicamente el reensamblamiento de paquetes. Esta funcionalidad es utilizada por los protocolos de la capa de aplicación para identificar sesiones que podrían ser ignoradas y descargar

información de identificación relativa a la sesión (protocolo de aplicación, dirección…) que puede ser usada posteriormente por las reglas del Detector Engine.

A su vez Stream5 es capaz de generar 8 alarmas distintas (mediante el generador ID 129) relacionadas con las anomalías TCP:

1. SYN de establecimiento de sesión.
2. Datos de un paquete SYN.
3. Datos enviados a un stream que no acepta datos.
4. El TCP Timestamp está fuera de la PAWS Windows.
5. Segmento dañino: overlap adjusted size menor o igual que 0.
6. Tamaño de la ventana menor que el que permite la política.
7. Limita el número de “overlaping TCP packets” recibidos.
8. Datos enviados después del paquete Reset.

3.3. SfPortscan

Diseñado por *Sourcefire,* el objetivo de este módulo es detectar la primera fase de un ataque: el *Reconocimiento*.

En esta fase el atacante intenta determinar qué protocolos de red o servicios soporta el host a atacar, para ello es usual realizar un escaneo de puertos (Portscan). Como se supone que el atacante tiene pocos conocimientos del sistema que desea atacar, este realizará un escaneo de puestos “a ciegas” y por tanto muchos mensajes de error serán enviados al atacante por parte de la víctima. Este comportamiento no es habitual en comunicaciones legítimas, y *sfPortscan* aprovecha esta característica para generar las alertas.

El preprocesador *sfPortscan* es capaz de detectar los siguientes ataques:

1. TCP/UDP/IP Portscan. El host atacante escanea múltiples puertos del host víctima.

2. TCP/UDP/IP Decoy Portscan. El atacante tiene un sistema que intenta camuflar su dirección IP con multiples direcciones.

3. TCP/UDP/IP Distributed Portscan. Múltiples host´s atacantes escanean un único host víctima.

4. TCP/UDP/IP/ICMP Portswap. Un único host escanea un único puerto del host víctima. Esto suele ocurrir cuando un nuevo exploit sale a la luz y un atacante intenta aprovecharse de él.

SfPortscan también es capaz de detectar estos ataques cuando son filtrados (Filters Alerts). Esto ocurre cuando el host víctima no emite mensajes de error al host atacante (pero no porque no debiera enviarlos, sino porque está configurado para no enviar nunca mensajes de error).

3.4. RPC Decode

Normaliza múltiples registros fragmentados RCP en un único registro. Este proceso de normalización de paquetes se realiza mediante un buffer de paquetes.

3.5. Performance Monitor

Este preprocesador mide a Snort en tiempo real y teoriza su máximo rendimiento, es decir crea estadísticas sobre la ejecución actual de Snort. Tiene dos modos de ejecución. En el modo *console* las estadísticas se mostrarán por consola. En el modo *file* las estadísticas se almacenarán en un fichero cuyo nombre tendrá que ser especificado.

3.6. HTTP Inspect

HTTP Inspect es un decodificador genérico de HTTP para aplicaciones de usuario. Dado un buffer, este preprocesador lo decodificará, buscará los campos HTTP y los normalizará. Normaliza tanto las respuestas del cliente como las respuestas del servidor.

HTTP Inspect trabaja buscando los campos HTTP de paquete en paquete, esto quiere decir que será engañado si los paquetes están fragmentados. Por ello, funciona de manera excelente de manera conjunta con otro módulo que vaya reensamblando los fragmentos HTTP previamente (por ejemplo el Stream5).

3.7. SMTP Preprocessor

SMTP Preprocesor es un decodificador SMTP para aplicaciones de usuario. Dado un buffer, este preprocesador lo decodificará y buscará los comandos y las respuestas SMTP. Además marcará los comandos, las secciones de datos de la cabecera y del cuerpo y los datos TLS.

3.8. FTP/Telnet Preprocessor

FTP/Teltnet es una mejora del decodificador Telnet y proporciona la capacidad de inspeccionar tanto los flujos de datos de FTP como los flujos de datos Telnet. Decodifica el flujo, los comandos y respuestas FTP, y las secuencias de escape y los campos normalizados de Telnet.

3.9. SSH

Este preprocesador detecta los siguientes exploits que afectan al protocolo SSH: Gobbles, CRC 32, Secure CRT y Protocol Mismatch.

Los ataques a los exploits Gobbles y CRC 32 ocurren después del intercambio de claves. Ambos ataques envían grandes payloads al servidor

inmediatamente después de que la autenticación haya ocurrido. Por ello, para detectar ambos exploits el *preprocesador SSH* cuenta el número de bits transmitidos al servidor, si dicho número de bytes excede un número predefinido dentro de un número determinado de paquetes, entonces se genera una alerta.

Los exploits Secure CRT y Protocol Mismatch son observables antes de que tenga lugar el intercambio de claves.

3.10. DCE/RPC

El preprocesador Dce/Rpc detecta y decodifica el tráfico SMB y el tráfico DCE/RPC. Aunque principalmente está diseñado para dar preferencia a las peticiones DCE/RPC y únicamente decodifica el SMB para ser consciente de las peticiones DCE/RPC potenciales que conlleva el tráfico SMB.

Generalmente, las reglas de Snort pueden ser evadidas usando fragmentación en las capas SMB y TCP, por ello, con este preprocesador activado, a las reglas se les suministran los datos DCE/RPC reensamblados para que sean examinados.

Sin embargo, en la capa SMB, solamente se reensamblan los paquetes que fueron fragmentados usando WiteAndX (este problema será solucionado en futuras versiones).

3.11. DNS

Este preprocesador decodifica las respuestas DNS (*) y puede detectar los siguientes exploits: DNS Client RData Overflow, Obsolete Record Types y Experimental Record Types.

3.12. SSL/TLS

El tráfico encriptado debería ser ignorado por Snort por motivos de rendimiento y para reducir el número de falsos positivos. El preprocesador SSL/TLS decodifica el tráfico SSL y TLS y opcionalmente determina si Snort debería detenerse a inspeccionar dicho tráfico y cuando debería hacerlo.

Por defecto, el preprocesador SSL/TLS busca el handshake packet que sigue al tráfico encriptado que viaja en ambas direcciones. Si uno de los lados responde con una indicación de que algo ha fallado, entonces la sesión no es marcada como encriptada.

Para verificar que el tráfico encriptado enviado desde las dos partes es legítimo es necesario asegurarse de dos cosas: que el último handshake packet no fue manipulado para evadir a Snort y que el tráfico es legítimamente encriptado.

En la mayoría de los casos, especialmente cuando los paquetes pueden perderse la única respuesta que se observa del endpoint son los TCP ACK´s (*trama del protocolo TCP que se utiliza como acuse de recibo*).

3.13. ARP Spoof Preprocessor

El preprocesador ARP Spoof Preprocessor decodifica los paquetes ARP y detecta los ataques ARP, las peticiones unicast ARP y los mapeos inconsistentes de Ethernet a IP.

3.14. DCE/RPC 2 Preprocessor

El propósito principal de este preprocesador es tratar la segmentación SMB y la segmentación DCE/RPC para anular las técnicas de evasión de detectores mediante segmentación.

4. MODOS DE EJECUCION DE SNORT

Snort puede configurarse para ejecutarse en 4 modos distintos de ejecución:

- Sniffer Mode.
- Packet Logger Mode.
- Network Intrusion Detection System (NIDS) Mode.
- Inline Mode.

A continuación, veremos las características y el funcionamiento de estos cuatro modos de ejecución.

4.1. Sniffer

Snort puede utilizarse como un sniffer del tráfico que pasa por una red. El programa lee los paquetes y los muestra por consola según los va capturando.

Algunas acciones que se pueden realizar en este modo son:

- Mostrar por consola las cabeceras de los paquetes TCP/IP

```
./snort -v
```

- Mostrar por consola las cabeceras de los paquetes TCP/IP, UDP, ICMP

```
./snort –vd
```

4.2. Packet Logger

Este modo de ejecución es similar al Sniffer Mode, con la diferencia de que en lugar de mostrar los paquetes capturados por consola, estos se almacenan en disco.

Algunas de las acciones que se pueden realizar en este modo son:

- Registrar los paquetes en la carpeta */log* (se supone que existe una carpeta llamada log en el directorio actual, en caso contrario la ejecución del comando dará un mensaje de error. También se puede especificar la ruta absoluta de la carpeta)

```
./snort -dev -l ./log
```

- Si solamente queremos registrar los paquetes de una red en particular debemos especificar su dirección poniendo la opción -h:

```
./snort -dev -l ./log -h 192.168.1.0/24
```

- Si queremos mostrar por pantalla alguno de estos paquetes que hemos registrado (siendo packet.log el nombre del paquete que queremos mostrar por consola) usamos el comando:

```
./snort -dv -r packet.log
```

4.3. Network Intrusion Detection System

Este modo captura el tráfico de la red y lo analiza para detectar intrusiones. Para ello compara el tráfico con una serie de reglas y registra los paquetes que hagan saltar alguna alarma, es decir, que correspondan con algún comportamiento sospechoso previamente definido. Es el modo que vamos a usar en nuestro sistema híbrido.

Para lanzar este modo de ejecución usamos el siguiente comando:

```
./snort -dev -l ./log -h 192.168.1.0/24 -c snort.conf
```

Siendo

- *.log* la carpeta especificada para registrar los paquetes
- *192.168.1.0/24* la dirección de la red que queremos proteger
- *Snort.conf* el fichero de configuración de Snort (que contiene, entre otras, información relativa a nuestras reglas y a los preprocesadores que estamos usando). Si este fichero no se encuentra en el directorio en el que estamos trabajando debemos especificar su ruta absoluta dentro del sistema de ficheros.

En el modo NIDS podemos seleccionar 6 formatos distintos para los registros: fast, full, unsock, none, console, cmg. Más adelante veremos con más detenimiento las características de cada formato. Para usar un formato determinado usamos la opción –A. Por ejemplo, para utilizar el modo fast, introduciríamos por consola el comando:

./snort –A fast -dev -l ./log -h 192.168.1.0/24 -c snort.conf

Por defecto, cuando ejecutamos el modo NIDS, primero se aplican las reglas Alert, después las reglas Pass y por último las reglas Log.

4.4. Modo Inline

En este modo de ejecución de Snort, se obtienen los paquetes de los Iptables (reglas sobre los diferentes paquetes de una red usadas por el administrador del sistema) y se usan nuevas reglas para determinar si se permite o no el paso de esos paquetes. Es decir, los Iptables utilizarán las reglas de Snort para determinar el paso de los paquetes.

Existen tres tipos de reglas en este modo:

- Drop. Los paquetes se dejan pasar o no y se registran mediante los mecanismos usuales de Snort.

- Reject. Se realizan las mismas acciones que con las reglas Drop y además se envía un *TCP reset* si el protocolo es el TCP o un *ICMP port unreachable* si el protocolo es el UDP.

- Sdrop. Se deja pasar o no a los paquetes, pero no se registra ninguno.

El Inline Mode es un modo de ejecución ligeramente diferente a los vistos hasta ahora (en un principio era una herramienta adicional que usaba como base Snort). Debido a esto, para poder ejecutarlo la instalación de Snort debe hacerse de la siguiente manera:

```
./configure --enable-inline

make

make install
```

Además, debemos asegurarnos de que el módulo ip_queue está cargado en nuestro sistema. Para ello podemos utilizar el siguiente comando:

```
iptables -A OUTPUT -p tcp --dport 80 -j QUEUE
```

Ahora ya podríamos utilizar Snort en modo inline mediante el siguiente comando:

```
snort_inline -QDc ../etc/drop.conf -l /var/log/snort
```

5. ALERTAS

5.1. Fichero de alertas generado por Snort

El fichero de alertas de Snort (*Alerts.ids*) tiene un formato ASCII plano, por lo que se puede visualizar fácilmente mediante cualquier procesador de textos.

Por ejemplo, imaginemos que ejecutamos Snort en modo NIDS y a su vez seleccionados para generar las alertas el modo fast, para ello introduciríamos por consola el comando:

./snort –A fast -dev -l ./log -h 192.168.1.0/24 -c snort.conf

Al ejecutar este comando Snort generará dos ficheros: uno llamado *Alerts.ids* (que contendrá información básica relativa al ataque) y otro fichero llamado Snort.log.xxxxxx (en el que se registrarán los paquetes que causaron las alarmas). Estos ficheros se almacenarán en el directorio */log* que a su vez se encuentra en el directorio en el que realizamos la instalación de Snort.

Ahora vamos a ver cómo sería el "núcleo" de las alertas que genera Snort:

[**] [116:56:1] (snort_decoder): T/TCP Detected [**]

- El primer número (116) es el Generator ID, que indica qué componente de Snort generó la alerta.

- El segundo número (56) es el Snort ID, que indica qué regla generó la alerta.

- El tercer número (1) en el Revision ID, que indica el número de veces que se violó dicha regla.

5.2. Formato de las alertas

Snort dispone de 11 modos distintos para generar las alertas. La elección de un formato u otro depende de los detalles que necesitemos o de cómo deseemos visualizar esas alertas.

La configuración de los módulos de salida se hace con el archivo de configuración snort.conf. Al abrir este fichero tenemos que buscar la sección output plug-ins. Por defecto esta sección está etiquetada con el comentario *Step #3: Configure output plug-ins.* Los distintos modos que podemos usar son:

1. alert_syslog. Envía las alarmas al syslog. De esta manera si una computadora es atacada los log´s no se ven comprometidos, puesto que fueron mandados a otro sistema.

2. alert_fast. Se escriben las alertas en formato de una línea en el fichero que especifique el usuario. Es un modo muy rápido puesto que Snort no desperdicia capacidad de procesamiento haciendo conversiones de formato.

Para utilizar este modo deberemos escribir en el fichero snort.config la siguiente línea:

output alert_fast: snort.log

Para utilizar este modo al ejecutar Snort por consola debemos añadir la opción –A fast.

Las alertas generados en este formato contienen la siguiente información: fecha y hora, mensaje de la alerta (núcleo), clasificación, prioridad de la alerta, protocolo, IP y puertos de origen y destino.

```
01/16-22:57:23.872383  [**] [1:1122:4] WEB-MISC /etc/passwd [**]
               [Classification: Attempted Information Leak] [Priority: 2] {TCP}
               192.168.1.68:44258 -> 172.16.34.18:80
```

3. alert_full. Si seleccionamos el modo Full obtendremos, además de la información del formato Fast, las cabeceras de los paquetes que

generan alertas. Este modo puede hacer que Snort no analice todos los paquetes en redes con mucho tráfico al no poder procesarlos. Debido a esto es un modo poco recomendable incluso en redes con poco tráfico.

La sintaxis a utilizar en el fichero snort.config para utilizar el modo alert_full es la siguiente:

output alert_full: alert.full

Para utilizar este modo al ejecutar Snort por consola debemos añadir la opción *–A full.*

En el siguiente ejemplo podemos ver cómo se mostrarán los ataques en este formato:

```
[**] [1:1201:7] ATTACK-RESPONSES 403 Forbidden [**]
[Classification: Attempted Information Leak] [Priority: 2]
01/16-23:06:11.675864 0:A0:CC:D2:10:31 -> 0:10:67:0:B2:50 type:0x800 len:0x268
172.16.34.18:80 -> 192.168.1.68:44561 TCP TTL:64 TOS:0x0 ID:12687 IpLen:20
               DgmLen:602 DF
***AP*** Seq: 0x2B34D8BB  Ack: 0xD53497B1  Win: 0x16A0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 135653685 322913730
```

4. alert_unixsock. Manda las alertas a través de un socket para que las escuche otra aplicación. Está opción solo puede usarse en Linux/UNIX.

5. log_tcpdump. Se registran los datos en formato binario (tcpdump-formatted file). Esto permite que Snort se ejecute más rápido (no se pierde tiempo de conversión de datos). Este modo es especialmente útil cuando deseamos hacer un análisis posterior del tráfico de una red.

Con los datos en formato tcpdump podemos utilizar cualquier utilidad de procesamiento. Para usar este modo de alertas escribiremos en el archivo de configuración la siguiente línea:

output log_tcpdump: snort.dump

Si se desea registrar todos los paquetes debe usarse la opción –b. Con la opción-L especificamos el destino:

/usr/local/bin/snort –b –L /var/log/snort/snort.dump

6. Database. Este modo guarda los log´s en una base de datos con las ventajas que ello conlleva: mayor velocidad y facilidad para buscar los archivos.

La base de datos se puede configurar de manera que solamente almacene alertas, log´s o las dos cosas. Snort solo soporta bases de datos que usen MySQL.

Para usar este modo incluiremos en el archivo de configuración la siguiente línea:

output database: <log | alert>, <database type>, <parameter list>

Los parámetros que deberemos incluir en el fichero de configuración snort.conf se especifican con el par *name=value y son los siguientes:*

- Host. Nombre o dirección IP del host el que se está ejecutando el servidor de la base de datos. Si la base de datos está en el mismo sistema donde está Snort el nombre del host será *localhost*.
- Port. Puerto de escucha del servidor de la base de datos.
- Password. Contraseña con la que se accede a la base de datos (opcional)
- Dbname. Nombre de la base de datos.
- Encoding. Método de codificación de los paquetes (Hex, Base64 o Ascii).
- Sensor_name. Nombre del sensor de Snort (generalmente no es necesario especificarlo
- Detail. Nivel de detalle con el que se almacenan los paquetes (full o fast)

Una posible configuración podría ser:

output database: log, mysql, user=snortuser password=h4wg dbname=snortdb host=localhost

7. CSV. CSV es un formato intermedio universal para exportar datos en una base de datos. Se utiliza cuando se está utilizando una base de datos que no es soportada por Snort.

8. Unified. Este modo está diseñado para ser uno de los más rápidos, de manera que los log se registran en formato binario.

9. Unified2. Unified2 es una versión ampliada del modo Unified. Pudiendo ejecutarse en 3 modos distintos:

- packet logging
- alert logging
- true unified logging

10. alert_prelude. Este modo de ejecución se utiliza para registrar los log en una base de datos de Prelude (para obtener información sobre Prelude visitar la página web http://www.prelude-ids.org/).

11. log_null. Usando este modo de ejecución se desactivan las alarmas. Para lanzarlo usamos el siguiente comando:

```
snort -A none -c snort.conf
```

6. REGLAS

Para realizar la detección de las intrusiones Snort usa un conjunto de reglas que el usuario deberá guardar en la carpeta de instalación del programa. En la página oficial de Snort existen conjuntos de reglas más o menos actualizadas que pueden usarse. Sin embargo, los usuarios también pueden crear patrones que se ajusten mejor a la configuración de su red para conseguir una buena protección.

Podemos dividir las reglas de Snort en dos partes: la cabecera y las opciones.

6.1. Cabecera

La cabecera contiene la siguiente información:

- **Acción de la regla.** Hay 8 tipos distintos de acciones posibles:
  - Alert. Genera una alerta y después registra el paquete.
  - Log. Registra el paquete.
  - Pass. Ignora el paquete
  - Activate. Genera una alerta y después llama a una regla dinámica.
  - Dynamic. Se activa cuando es llamada por Activate, después se comporta como una regla de tipo log.
  - Drop. Pasa el paquete por los iptables y después lo registra.
  - Reject. Se realizan las mismas acciones que con las reglas Drop, y además se envía un paquete *TCP reset* si el protocolo es TCP o un paquete *ICMP port unreachable* si el protocolo es UDP.
  - Sdrop. Pasa el paquete por los iptables pero no lo registra.
- **Protocolo.** Los protocolos que analiza Snort son TCP, UDP, ICMP e IP.

➢ **Dirección IP Origen.** Al introducir la dirección IP también tenemos que introducir la clase de dicha dirección, por ejemplo, para una dirección de clase C deberemos poner *192.168.1.0/24*.

➢ **Puerto IP Origen.** Al indicar el puerto es posible que queramos utilizar un rango determinado de puertos, en los siguiente ejemplos explicaremos las posibilidades:

- 1:1024 = del puerto 1 al puerto 1024 (ambos inclusive)
- :1024 = todos los puertos menores o igual a 1024
- 1024: = todos los puertos mayores o igual que 1024
- !10:1024 = del puerto -10 al puerto 1024 (ambos inclusive).

➢ **Dirección IP Destino.** Las condiciones son iguales que con la dirección IP origen.

➢ **Puerto IP Destino.** Las condiciones son iguales que con el puerto IP origen.

➢ **Dirección de la operación.** Este operador indica la dirección del tráfico al que se aplica la regla. Puede ser -> (saliente), <- (entrante) o <> (bidireccional).

6.2. Opciones

Podemos clasificar las opciones de una regla en 4 categorías:

6.2.1. Opciones generales

Las opciones generales de una regla son:

1. **Msg.** Indica el mensaje a mostrar cuando se genera la alerta.
2. **Reference.** Permite incluir referencias a sistemas de identificación de ataques externos.

3. Gid (Generator ID). Se utiliza para identificar qué elemento de Snort generará el evento cuando una regla se dispara.

4. Sid. Se utiliza para identificar las reglas de Snort.

- <100 se reserva para uso futuro.
- 100-1.000.000 reglas incluidas en la distribución de Snort
- >1.000.000 reglas locales.

5. Rev. Se utiliza para indicar la versión de la regla.

6. Classtype. Se utiliza para organizar las reglas en función del tipo de ataque que detecten. En el cuadro 2.1 pueden verse todos los tipos y sus características.

7. Priority. Indica el nivel de prioridad de la regla.

8. Metadata. Permite al escritor de la reglas añadir información adicional sobre la regla.

Tipo de clase	Descripción	Prioridad
attempted-admin	Intento de obtención de privilegios de administrador	Alta
attempted-user	Intento de obtención de privilegios de usuario	Alta
kickass-porn	Contenido pornográfico	Alta
policy-violation	Violación de la política de privacidad de la empresa	Alta
shellcode-detect	Código ejecutable detectado	Alta
successful-admin	Obtención de privilegios de administrador	Alta
successful-user	Obtención de privilegios de usuario	Alta
trojan-activity	Troyano detectado	Alta
unsuccessful-user	Intento no exitoso de obtención de privilegios de usuario	Alta
web-application-attack	Web Application Attack	Alta
attempted-dos	Intento de Denial of Service	Media
attempted-recon	Intento de escape de información	Media
bad-unknown	Potentially Bad Traffic	Media
default-login-attempt	Intento de login con un usuario y una contraseña por defecto	Media
denial-of-service	Detección de un ataque de Denial of Service	Media
misc-attack	Misc Attack	Media
non-standard-protocol	Detección de un protocolo no estándar	Media
rpc-portmap-decode	Decodifica un RCP Query	Media
successful-dos	Denial of Service exitoso	Media
successful-recon-largescale	Large Scale Information Leak exitoso	Media
successful-recon-limited	Information Leak exitoso	Media
suspicious-filename-detect	Fichero sospechoso detectado	Media
suspicious-login	Intento de login utilizando un nombre de usuario sospechoso	Media
system-call-detect	Llamada al sistema detectada	Media
unusual-client-port-connection	Conexión de un cliente utilizando un puerto inusual	Media
web-application-activity	Acceso a una aplicación Web potencialmente vulnerable	Media
icmp-event	Evento genérico ICMP	Baja
misc-activity	Misc activity	Baja
network-scan	Detección de un escaneo	Baja
not-suspicious	Tráfico no sospechoso	Baja
protocol-command-decode	Generic Protocol Command Decode	Baja
string-detect	Detección de un string sospechoso	Baja
Unknown	Tráfico desconocido	Baja
tcp-connection	Intento de conexión TCP detectado	Muy baja

Tabla 2.1. Tipos de clases de las reglas de Snort

6.2.2. Opciones de Payload

Las opciones de payload son las siguientes:

1. content. Permite buscar en el payload un contenido específico y realizar la respuesta correspondiente a dicho contenido.

2. nocase. Indica que en el caso de que Snort esté buscando un determinado patrón, lo ignore.

3. rawbytes. Permite a la regla mirar los datos previos a la decodificación que realizaron los preprocesadores.

4. depth. Determina la cantidad de datos a analizar del paquete (no es obligatorio analizar el paquete en su totalidad).

5. offset. Especifica a partir de qué punto del paquete empieza la búsqueda (no es obligatorio analizar el paquete desde el principio).

6. distance. Indica qué cantidad de datos se ignoran del payload hasta que se comienza a realizar la búsqueda del patrón que especifica la regla.

7. within.

8. http_client_body. Restringe la búsqueda al cuerpo normalizado de un http client request.

9. http_cookie. Restringe la búsqueda al campo Cookie Header de un http client request.

10. http_header. Restringe la búsqueda al campo Header de un http client request.

11. http_method. Restringe la búsqueda al fragmento Method de un http client request.

12. http_uri. Restringe la búsqueda al campo NORMALIZED request URI.

13. fast_pattern. Modifica el contenido de las especificaciones de una regla para que puedan ser utilizadas con el Fast Patern Matches.

14. uricontent. Busca la petición normalizada del campo URI (normalized request URI field).

15. urilen. Especifica la longitud exacta, la longitud mínima, la longitud máxima o el rango del URI lengths a buscar.

16. isdataat. Verifica que el payload tiene los datos en una localización específica.

17. pcre. Permite a las reglas escribirse usando expresiones regulares compatibles con Perl.

18. byte_test. Compara el campo byte con un valor específico.

19. byte_jump. Permite escribir reglas para protocolos con longitud cifrada de forma trivial.

20. ftpbounce. Permite detectar FTP Bounce attacks.

21. ans1. Decodifica un paquete o una porción de un paquete en busca de varias codificaciones maliciosas.

22. cvs. Ayuda en la detección de: Bugtraq-10384, CVE-2004-0396 (Malformed Entry Modified and Unchanged flag insertion).

6.2.3. Opciones Non- Payload

Las opciones non-payload son las siguientes:

1. **fragoffset.** Permite comparar el campo IP Fragment Offset con un valor decimal.
2. **ttl.** Permite chequear el IP time-to-live.
3. **tos.** Permite chequear el campo IP TOS con un valor específico.
4. **id.** Permite chequear el campo IP ID con un valor específico.

5. **ipopts.** Permite chequear si una opción IP específica está presente.

6. **fragbits.** Permite chequear si la fragmentación y los bits de reserva están especificados en la cabecera IP.

7. **dsize.** Se utiliza para testear el tamaño del payload.

8. **flags.** Permite chequear si algún TCP flag está especificado.

9. **flow.** Se utiliza para asegurarse de la dirección del flujo del tráfico.

10. **flowbits.** Permite buscar huellas a través de sesiones del protocolo de transporte.

11. **seq.** Chequea el TCP sequence number.

12. **ack.** Chequea el TCP acknowledge number.

13. **window.** Chequea el tamaño de la ventana TCP.

14. **itype.** Chequea el ICMP type value.

15. **icode.** Chequea el ICMP code value.

16. **icmp_id.** Chequea el ICMP ID value.

17. **icmp_seq.** Chequea el ICMP secuence value.

18. **rpc.** Chequea la aplicación RPC, la versión y el número de procedimiento en una petición SUNRPC CALL.

19. **ip_proto.** Chequea la cabecera del protocolo IP.

20. **sameip.** Chequea si la dirección IP de la fuente es la misma que la dirección IP del destino.

21. **stream_size.** Permite comparar el tráfico en función del número de bytes analizados, como los que se determinan mediante los números de secuencia TCP. Esta opción solamente está disponible cuando el preprocesador Stream5 está activado.

6.2.4. Opciones post-detección

Las opciones post-detección son las siguientes:

1. logto. Indica que se registren todos los paquetes que disparen las alarmas de esta regla a un directorio especial de registro.

2. session. Se utiliza para extraer los datos de usuario de la sesiones TCP.

3. resp. Se utiliza para intentar cerrar una sesión cuando una alerta es disparada.

4. react. Se utiliza para reaccionar al tráfico que provocó que se disparase una alerta: se cierra la conexión y se manda una notificación.

5. tag. Permite registrar más información además del paquete que disparó la alerta.

6. activates. Permite que se añada una regla cuando la alerta se dispara.

7. activated_by. Permite activar dinámicamente una regla cuando la alerta se dispara

8. count. Se utiliza conjuntamente con la opción anterior. Permite indicar cuantos paquetes se analizan con la nueva regla creada después de haberse creado.

IV
PROPUESTA HÍBRIDA

1. ANÁLISIS FUNCIONAL

1.1. Definición del prototipo

El prototipo desarrollado es un IDS híbrido que combina detección por firmas y detección por anomalías. El sistema simultanea la ejecución de Snort con la ejecución de un detector de anomalías cuyo diseño se basa en la propuesta de Anagram.

El detector de anomalías está destinado a la detección de código malicioso dentro del tráfico de una red. Para capturar el tráfico se ha implementado un sniffer usando librerías pcap.

El funcionamiento del detector de anomalías tiene dos fases: entrenamiento y detección.

Fase de entrenamiento

La fase de entrenamiento sirve para crear un patrón que define el tráfico normal de una red y para definir los umbrales para distinguir tráfico legítimo de tráfico malicioso.

Esta fase se desarrolla en tres etapas:

Etapa 1

La primera etapa consiste en analizar un conjunto de paquetes de tráfico legítimo usado para rellenar el bloom filter. Estos paquetes se dividirán en 7-gram y de cada 7-gram se calculará su función hash MD5 para determinar su posición dentro del bloom filter. A continuación, se incrementa en 1 la posición determinada en el bloom filter auxiliar.

Una vez analizado todo el tráfico, se usa la información del bloom filter auxiliar para calcular la desviación estándar y rellenar el bloom filter binario que será el que se utilice durante la detección.

Etapa 2

La segunda etapa consiste en probar el detector con tráfico legítimo y calcular la media de n-grams sospechosos que se han encontrado en estos paquetes. Esta media servirá como cota inferior del umbral de n-grams sospechosos que se usará para detectar el código malicioso.

Etapa 3

La tercera etapa consiste en probar el detector con tráfico con ataques y calcular la media de n-grams sospechosos que se han encontrado en estos paquetes. Esta media servirá como cota superior del umbral de n-grams sospechosos que se usará para detectar el código malicioso.

Fase de detección

La fase de detección procesa los paquetes introducidos (ya sea leidos de un archivo o capturados de la red) y los compara con los datos almacenados en el bloom filter.

Para ello cada paquete se divide en 7-gram y se calcula la función hash MD5 de cada 7-gram. Una vez resumidos los paquetes (por medio de la función hash) se comprueba cuantos n-grams no aparecen en el bloom filter rellenado durante el entrenamiento. Si este número es mayor que el umbral fijado en la fase de entrenamiento de considerará que el paquete es sospechoso y se lanzará una alarma.

Alarmas

El prototipo diseñado genera dos tipos de alarmas. Por un lado, se crea un archivo con las alarmas generadas por Snort y por otro lado se crean dos archivos con las alarmas generadas por el detector de anomalías (un archivo pcap con los paquetes sospechosos y un fichero de texto con los datos sobre las alarmas).

1.2. Conceptos teóricos

En este apartado vamos a explicar algunos conceptos teóricos y estructuras utilizadas en el prototipo.

1.2.1. N-gram

Es una técnica utilizada en el procesamiento del leguaje natural. Su principal objetivo es la predicción, dentro de una sucesión, del siguiente elemento sabiendo los anteriores y las distintas probabilidades de aparición.

El funcionamiento en el ámbito de los IDS y, por lo tanto, de este prototipo es el siguente:

Se elige un cierto tamaño n (en nuestro caso 7) que determinará el número de bytes que formarán cada n-gram. El sistema actuará como una ventana de tamaño n por la que irán pasando los bytes del paquete de forma secuencial. Cada n-gram aparecido en un paquete, es decir, cada secuencia de n bytes consecutivos que tenga el paquete, se registrará en algún tipo de estructura. El contenido del paquete vendrá determinado por el registro de sus n-grams que se usará para realizar el patrón de comportamiento o para compararlo con éste dependiendo del modo de ejecución del sistema.

Esta técnica nos permite resumir el contenido de los paquetes y buscar dependencias y similitudes entre ellos. La principal ventaja de su uso es el poco consumo de recursos y facilidad para implementarla. Por otro lado, el inconveniente es la generación de muchos n-grams lo que requiere alguna técnica auxiliar que permita resumir la información.

1.2.2. Bloom filter

Un bloom filter es una estructura de datos cuyo objetivo es almacenar la información sobre los n-grams que aparecen en un paquete. Podemos

imaginarnos el bloom filter como un array en el que cada posición representa un tipo de n-gram.

En la práctica, un bloomfilter es un array de bits en el que un 0 representa que ese n-gram no aparece en el tráfico normal de la red y un 1 representa que ese n-gram si se ha encontrado anteriormente. Al usar bits para representar n-grams conseguimos reducir en gran medida la memoria necesaria para guardar la información sobre el tráfico legítimo.

El uso del bloom filter es distinto en cada fase del sistema. En la fase de entrenamiento se rellena el bloom filter con los n-gram que aparecen en el tráfico legítimo. En la fase de detección se usa para comprobar si los n-gram que van apareciendo en el tráfico a analizar pertenecen a tráfico legítimo o pueden ser ataques.

Un problema del uso del bloom-filter es la gran cantidad de n-gram distintos que puede haber en el tráfico de entrenamiento, lo que requiere un número de posiciones en el array escesivo. Para solucionarlo se utiliza una función hash que resume el contenido de cada n-gram y limita el número de posibilidades. Esta función hash debe garantizar las mínimas colisiones entre elementos.

1.2.3. Función Hash: MD5

Para realizar el resumen de los n-gram y elegir la posición del bloom filter que representa a cada n-gram relizamos la función hash MD5 con el contenido de cada n-gram. El algoritmo de esta función hash consta de cinco pasos explicados a continuación:

1. Añadir bits de relleno

El mensaje será extendido hasta que su longitud en bits sea congruente con 448, módulo 512. Es decir, se añaden bits al mensaje hasta que su longitud diste 64 de ser un múltiplo de 512. Esta extensión se realiza siempre, incluso si la longitud del mensaje es ya congruente con 448, módulo 512.

La extensión se realiza como sigue: se añade un sólo bit "1" al mensaje, y después se añaden bits "0" hasta que la longitud en bits del mensaje extendido se haga congruente con 448, módulo 512. En todos los mensajes se añade al menos un bit y como máximo 512.

2. Longitud del mensaje

Un entero de 64 bits que representa la longitud 'b' del mensaje (longitud antes de añadir los bits) se concatena al resultado del paso anterior. En caso de que 'b' sea mayor que 2^64 sólo se usarán los 64 bits de menor peso de 'b'.

En este punto, el mensaje (después de rellenar con los bits y con 'b') tiene una longitud que es un múltiplo exacto de 512 bits. A su vez, la longitud del mensaje es múltiplo de 16 palabras (32 bits por palabra). Con M[0 ... N-1] denotaremos las palabras del mensaje resultante, donde N es múltiplo de 16.

3. Inicializar el buffer MD

Un búfer de cuatro palabras (A, B, C, D) se usa para calcular el resumen del mensaje. Aquí cada una de las letras A, B, C, D representa un registro de 32 bits. Estos registros se inicializan con los siguientes valores hexadecimales, los bits de menor peso primero:

palabra A: 01 23 45 67

palabra B: 89 ab cd ef

palabra C: fe dc ba 98

palabra D: 76 54 32 10

4. Procesado del mensaje e bloques de 16 palabras

Primero definimos cuatro funciones auxiliares que toman como entrada tres palabras de 32 bits y cuya salida es una palabra de 32 bits.

En cada posición de cada bit F actúa como un condicional: si X, entonces Y sino Z. La función F podría haber sido definida usando + en lugar de v ya que XY y not(x) Z nunca tendrán unos ('1') en la misma

posición de bit. Es interesante resaltar que si los bits de X, Y y Z son independientes y no sesgados, cada uno de los bits de F(X,Y,Z) será independiente y no sesgado.

Las funciones G, H e I son similares a la función F, ya que actúan "bit a bit en paralelo" para producir sus salidas de los bits de X, Y y Z, en la medida que si cada bit correspondiente de X, Y y Z son independientes y no sesgados, entonces cada bit de G(X,Y,Z), H(X,Y,Z) e I(X,Y,Z) serán independientes y no sesgados. Nótese que la función H es la comparación bit a bit "xor" o función "paridad" de sus entradas.

Este paso usa una tabla de 64 elementos T[1 ... 64] construida con la función seno. Denotaremos por T[i] el elemento i-ésimo de esta tabla, que será igual a la parte entera del valor absoluto del seno de 'i' 4294967296 veces, donde 'i' está en radianes.

Paso 5. Salida

El resumen del mensaje es la salida producida por A, B, C y D. Se comienza con el byte de menor peso de A y se acaba con el byte de mayor peso de D.

Después de estos 5 pasos obtenemos un número de 128 bits que identifica el contenido del n-gram. Para seleccionar una posición del bloom filter se podemos coger un número determinado de bits que tengan suficiente rango para abarcar todas las posiciones del array.

1.2.4. Desviación estándar

Debido a la gran cantidad de n-grams que se generan durante el entrenamiento el bloom filter solía llenarse casi al completo impidiendo el correcto funcionamiento de prototipo. Para evitar este problema decidimos usar la desviación estándar del número de n-grams de cada tipo para eliminar del bloom filter aquellos menos importantes o representativos.

La desviación estándar es una medida del grado de dispersión de los datos con respecto al valor promedio. Dicho de otra manera, la desviación estándar es simplemente el "promedio" o variación esperada con respecto a la media aritmética.

Si los datasets utilizados para el entrenamiento son representativos del tráfico normal de la red conseguiremos lo que estadísticamente se denomina una distribución normal que se corresponderá con la gráfica de la figura 3.1. En dicha gráfica podemos observar como mas del 99% de los datos se encuentra dentro del intervalo [-3σ, 3 σ]

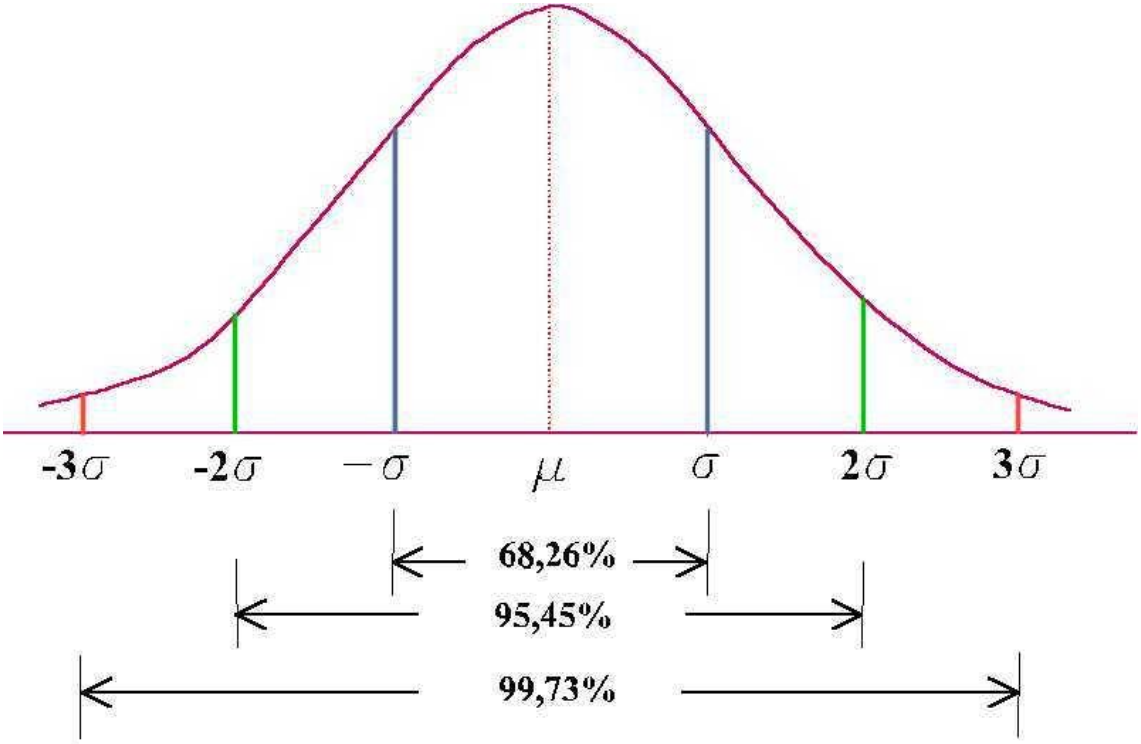

Figura 3.1. Distribución normal

Para poder calcular la desviación estándar usaremos un bloom filter auxiliar que guardará en cada posición el número de veces que ha aparecido el n-gram al que representa. Una vez rellando este bloom filter con los datos del tráfico de entrenameinto se seguirá el siguiente proceso:

1. Cálculo de la media de accesos a posición del bloom filter auxiliar.
2. Cálculo de la varianza mediante la siguiente fórmula:

$$\sigma^2 = \frac{\sum_{i=1}^{N} (X_i - \mu)^2}{N}$$

3. Cálculo de la desviación típica mediante la siguiente fórmula:

$$\sqrt{\sigma^2} = \sqrt{\frac{\sum_{i=1}^{N} (X_i - \mu)^2}{N}}$$

Una vez calculadas la varianza y la desviación típica podemos calcular el intervalo de valores representativo de la muestra y rellenar el bloom filter con los n-gram que estén dentro de este intervalo.

Estos datos estadísticos también serán utilizados para calcular el número de n-gram sospechosos con los cuales se considerará si un paquete pertenece a un ataque o no.

2. DISEÑO

2.1. Arquitectura

La arquitectura de nuestra propuesta puede verse en la figura 3.2. El sistema está dividido en diferentes módulos que se relacionan entre sí como se muestra en el esquema. Estos módulos son: Inicio, Sniffer, NidsReglas, Análisis y Alertas. A continuación explicamos la funcionalidad de estos módulos.

2.1.1. Inicio

Se encarga de reconocer la línea de comandos introducida por el usuario en el terminal, identificar las opciones seleccionadas y lanzar lo módulos correspondientes. Básicamente tiene que elegir que módulos lanzar dependiendo del modo de ejecución seleccionado.

Los modos de ejecución son los siguientes:

- **Entrenamiento**
  En el modo de entrenamiento se lanzará el módulo Sniffer en cualquiera de los 3 submodos de entrenamiento.
- **Pruebas**
  En el modo pruebas se lanza tanto el módulo Sniffer como el módulo NidsReglas
- **Detección**
  En el modo de detección se lanzan los módulos Sniffer y NidsReglas.

2.1.2. Sniffer

Este módulo se encarga de enviar los paquetes a analizar de uno en uno al módulo Análisis. Si estamos ejecutando en modo *Entrenamiento* o en modo *Pruebas* cargará los paquetes de un archivo pcap que indique el

usuario. Si estamos ejecutando en modo “Detección” se encargará de sniffear los paquetes de red por la interfaz indicada.

Además, si nos encontramos en los modos de ejecución “Pruebas” o “Detección” esperará al valor devuelto por el módulo de Análisis para registrar los paquetes en caso de producirse una alerta.

2.1.3. NidsReglas

Lanza una instancia de Snort para analizar el tráfico de forma paralela al detector de anomalías. De esta forma conseguimos analizar el tráfico usando las reglas de Snort para buscar ataques conocidos.

En caso de que Snort encuentre un ataque crea los archivos Snort.log y Alerts.ids para registrar los datos de los paquetes sospechosos.

Este módulo no se lanza si estamos ejecutando el sistema en modo “Entrenamiento”.

2.1.4. Análisis

Se encarga de analizar los paquetes que recibe del módulo Sniffer. A continuación, detallamos qué acciones realiza de acuerdo con el modo de ejecución introducido:

- **Entrenamiento**

  El modo de entrenamiento se encarga de analizar el tráfico de entrenamiento y calcular los datos que definirán el tráfico normal de la red. Tiene 3 submodos de ejecución que se corresponden con las 3 etapas del entrenamiento:

  - **Normal.** En este submodo se rellena el bloom filter con los datos del tráfico de entrenamiento. Para ello, se rellena primero la estructura x-bloomfilter con el número de n-grams de cada tipo encontrados. A continuación se determinan las posiciones más representativas del tráfico y, finalmente, se rellena el bloom filter

poniendo a 1 estas posiciones. Por último, guarda el contenido del BloomFilter en el fichero Bloomfilter.txt para que pueda ser usado en el futuro.

- **Legítimo.** En este submodo se calcula la media de n-grams sospechosos que se encuentran dentro del tráfico legítimo. Se usará para determinar el umbral de n-grams sospechosos que determinen si un paquete puede ser un ataque. Guarda los resultados en el fichero MediaLegítimos.dat

- **Ataques.** En este submodo se calcula la media de n-grams sospechosos que se encuentran dentro del tráfico de entrenamiento con ataques. Se usará para determinar el umbral de n-grams sospechosos que determinen si un paquete puede ser un ataque. Guarda los resultados en el fichero MediaAtaques.dat.

➢ **Pruebas.**

El modo de pruebas se encarga de analizar el tráfico que se encuentra guardado en un fichero pcap. Se utiliza principalmente para probar el prototipo analizando tráfico conocido o para analizar tráfico previamente capturado. Para detectar ataques divide el contenido de los paquetes en n-grams y comprueba si cada uno de ellos se encuentra en el bloom filter. Si no es así y el número de n-grams sospechosos supera un cierto umbral se comunica a los módulos de Sniffer y Alarmas para que guarden el paquete y lancen una alarma respectivamente.

➢ **Detección**

El módulo de detección actúa igual que el de pruebas con la diferencia de que el tráfico a analizar se captura de la red mediante el interfaz de red indicado por el usuario.

2.1.5. Alertas

Este módulo se encarga de registrar las alertas que lanza el módulo de análisis en un archivo de texto. Para ello crea un archivo llamado *fecha*Alertas.txt (donde fecha es la fecha en que se ha encontrado la

alarma en formato día-mes-año) en el que se escribirá una línea por cada paquete sospechoso encontrado.

La información que se escribirá de cada paquete sospechoso será la siguiente:

- **Hora**
  Hora en la que saltó la alerta en formato *horas:minutos:segundos.*

- **Puntuación**
  Puntuación del paquete que ha hecho saltar la alarma. Esto es el número de n-grams sospechosos y el número de n-grams totales del paquete.

A continuación se muestra un ejemplo de un archivo de alertas generado durante las pruebas:

```
16:28:18 Alerta por anomalia - puntacion:   3 /  830
16:28:18 Alerta por anomalia - puntacion:   3 / 1500
16:28:18 Alerta por anomalia - puntacion:   3 / 1500
16:28:18 Alerta por anomalia - puntacion:   5 / 1500
16:28:18 Alerta por anomalia - puntacion:   3 / 1500
16:28:18 Alerta por anomalia - puntacion:   3 / 1500
16:28:18 Alerta por anomalia - puntacion:   3 / 1500
16:28:18 Alerta por anomalia - puntacion:   3 / 1500
16:28:18 Alerta por anomalia - puntacion:   3 / 1500
16:28:18 Alerta por anomalia - puntacion:   3 / 1500
16:28:18 Alerta por anomalia - puntacion:   4 / 1500
```

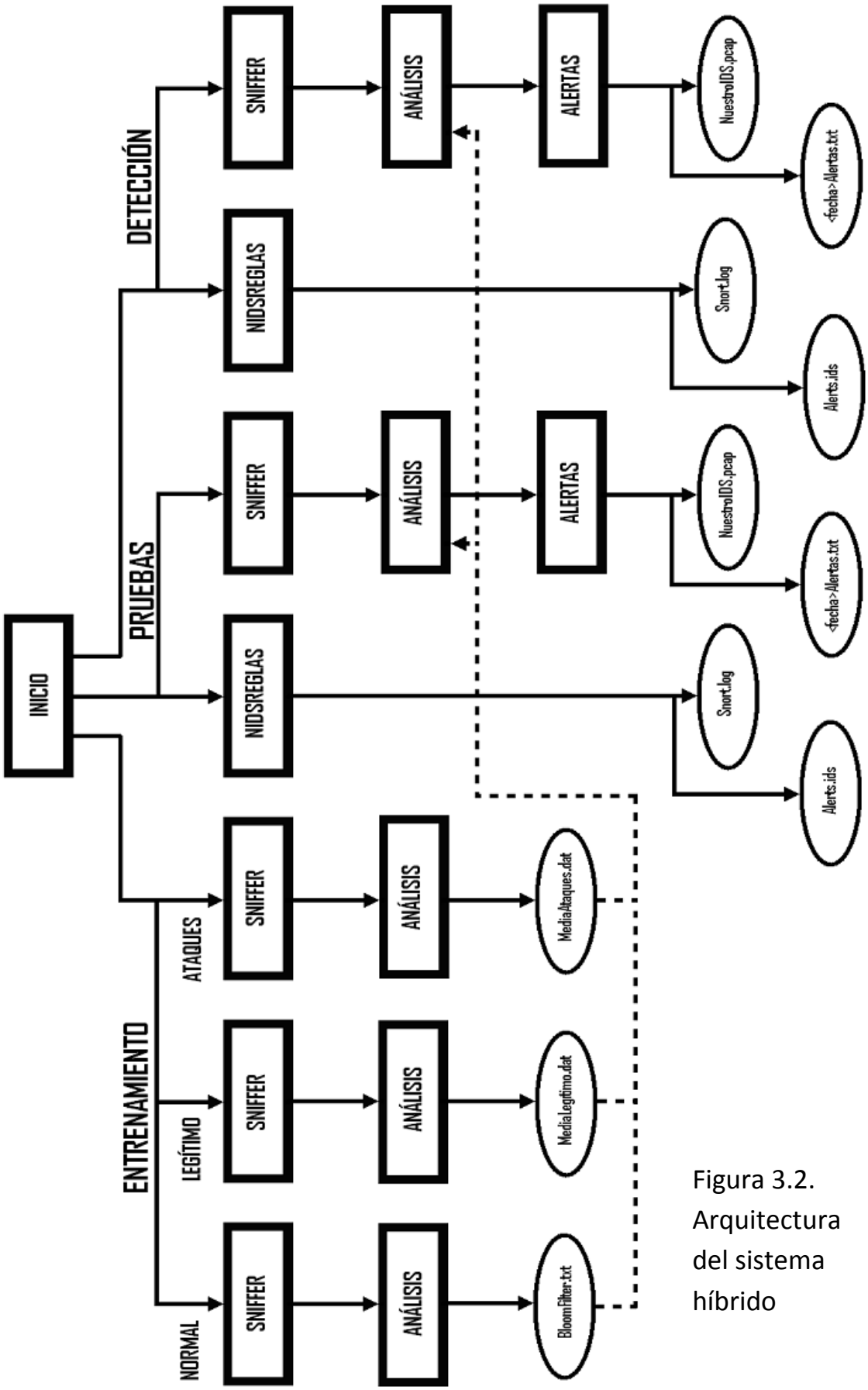

Figura 3.2. Arquitectura del sistema híbrido

2.2. Diagrama de clases

Aunque los diagramas de clases están diseñados para lenguajes orientados a objetos, nuestro código se ha implementado de forma modular por lo que hemos podido construir un diagrama que explica el diseño del código. Este diagrama puede verse en la figura 3.3.

A continuación explicaremos el objetivo y funcionamiento de cada uno de esos módulos.

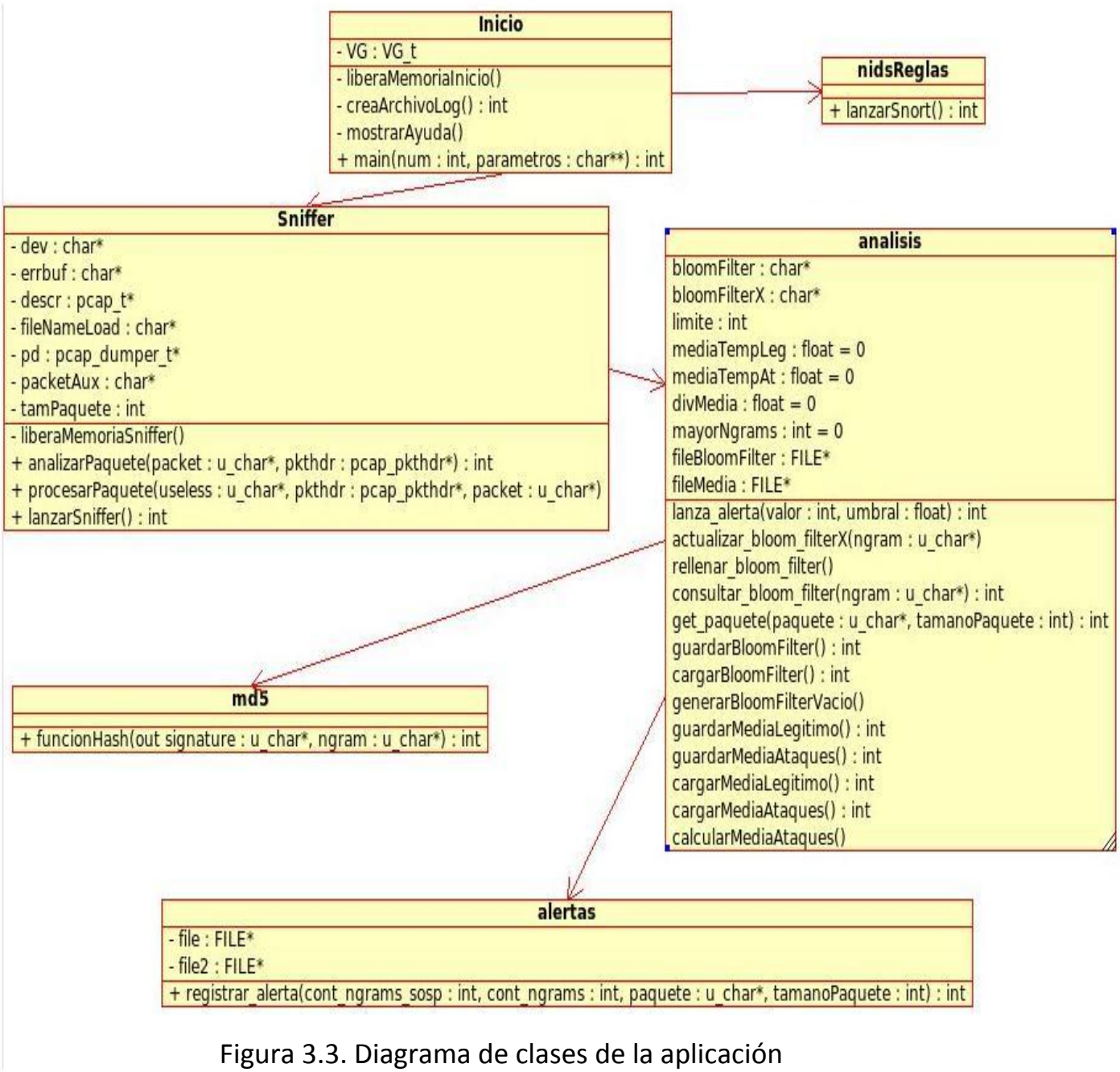

Figura 3.3. Diagrama de clases de la aplicación

2.2.1. Inicio

El módulo Inicio se encarga de leer el comando introducido por el usuario en el terminal y lanzar la ejecución del programa con las opciones elegidas. Para ello lee cada una de las opciones introducidas, comprueba que tanto las opciones como los datos son correctos y comunica a los módulos Sniffer y NidsReglas las características de la ejecución.

A continuación, explicamos la variable global y los métodos de este módulo:

- **VG**

  La variable VG es una estructura que contiene los datos con las opciones introducidas en el comando de ejecución, es decir, los modos de ejecución elegidos, las rutas de los directorios que se usarán, el interfaz elegido, etc.

- **liberaMemoriaInicio()**

  Este método libera la memoria ocupada por todas las variables creadas dentro de la estructura VG. Se usará al final de la ejecución del programa para liberar la memoria usada.

- **creaArchivoLog()**

  Este método crea el log que usará el detector de anomalías para registrar las alarmas generadas. Dicho archivo tendrá como nombre alertas.txt precedido por la fecha en que se ha ejecutado el programa. Si no puede crea el archivo devuelve error.

- **mostrarAyuda()**

  Este método muestra por consola el manual de uso del programa, es decir, las diferentes opciones que pueden usarse y la sintaxis que debe utilizarse.

- **main (int, char*)**

  Este método procesa el comando introducido para ejecutar el programa y separa los diferentes parámetros. Además comprueba si los datos son correctos y si lo son crea las variables que guardarán estos valores. Si algún valor no es correcto devolverá error. Una vez generadas todas las variables lanza la ejecución del programa.

2.2.2. NidsReglas

El módulo NidsReglas lanza la ejecución de Snort con las opciones elegidas por el usuario. El módulo tiene un único método:

- **lanzaSnort()**

  Este método se encarga ejecutar los comandos para la ejecución de Snort. En caso de que exista algún error se indicará por consola.

2.2.3. Sniffer

El módulo Sniffer se encarga de capturar el tráfico que será analizado por el detector de anomalías. El funcionamiento del Sniffer varía según el modo de ejecución del programa. Para el modo de entrenamiento y de pruebas los paquetes se recogerán del archivo especificado. Para el modo de detección los paquetes se recogen del interfaz de red especificado. Para realizar la captura de paquetes se utiliza la librería pcap.

Además, en los modos de prueba y detección guarda los paquetes que han generado alarma en un archivo pcap.

A continuación se explican todas las variables globales y los métodos del módulo:

- **liberaMemoriaSniffer ()**

Este método libera la memoria ocupada por todas las variables creadas dentro del módulo Sniffer. Se usará al final de la ejecución del programa para liberar la memoria usada.

- **analizarPaquete (const u_char*, const struct pcap_pkthdr*)**

Este método pasa el paquete al módulo de Análisis para que lo procese.

- **registrarPaquete (const u_char*, const struct pcap_pkthdr*)**

Este método sirve para guardar los paquetes analizados por el sistema en el archivo pcap introducido por parámetro.

- **procesarPaquete (u_char*, const struct pcap_pkthdr*, const u_char*)**

Este método comprueba si el paquete capturado es sospechoso (llamando a analizarPaquete) y si lo es lo registra en un archivo (llamando a registrarPaquete).

- **lanzarSniffer ()**

Este método se encarga de lanzar la ejecución del Sniffer para capturar paquetes. El método distingue entre los diferentes modos de ejecución para realizar las acciones propias de cada modo. Para la captura de los paquetes en la red se usa un bucle infinito, es decir, la captura no acabará hasta la finalización del programa.

2.2.4. Análisis

El módulo Análisis se encarga de procesar los paquetes capturados por el sniffer. Tiene diferentes funciones dependiendo del modo de ejecución.

En el modo de entrenamiento, una vez capturados todos los paquetes y calculado sus datos estadísticos se actualiza el bloom filter. En los modos de detección y prueba una vez analizados los paquetes se compararán con

el contenido del bloom filter y en caso de ser considerados peligrosos se notificará a los módulos Sniffer y Alertas para que creen los diferentes logs.

Las variables y métodos utilizados en este módulo son los siguientes:

- **int lanza_alerta (int valor, int umbral)**

Este método se encarga de pedir al módulo Alertas que lance una alarma cuando la puntuación de un paquete es superior al umbral determinado.

- **actualizar_bloom_filterX (const unsigned char* ngram)**

Obtiene el valor hash del ngram introducido y actualiza la posición del bloomfilter auxiliar sumándole uno.

- **rellenar_bloom_filter ()**

Utiliza los datos contenidos dentro del bloom filter auxiliar para calcular los datos estadísticos y rellenar el bloom filter que representa el patrón de comportamiento legítimo.

- **consultar_bloom_filter (const u_char* ngram)**

Este método obtiene el valor hash del n-gram introducido y consulta esa posición del bloom filter para determinar si el n-gram estaba en el tráfico legítimo.

- **get_paquete (u_char* paquete, int tamanoPaquete)**

Este método divide el paquete recibido en n-grams y procesa el paquete según el modo de ejecución en que esté funcionando. Es decir, llama a los métodos para hacer las operaciones requeridas en cada modo.

- **guardarBloomFilter ()**

Este método guarda el bloom filter en el archivo ./bloomfilter.txt para poder recuperarlo en otras ejecuciones.

- **cargarBloomFilter ()**

Este método carga el bloom filter del archivo ./bloomfilter.txt para poder usar un bloomfilter previamente entrenado.

- **generarBloomFilterVacio ()**

Este método inicializa el bloom filter a los valores por defecto, es decir, todos los valores a 0.

- **guardarMediaLegitimo ()**

Este método guarda en el archivo ./mediaLegitimo.dat la media de n-grams sospechosos encontrados en el entrenamiento con tráfico legítimo.

- **guardarMediaAtaques ()**

Este método guarda en el archivo ./mediaAtaques.dat la media de n-grams sospechosos encontrados en el entrenamiento con tráfico con ataques.

- **cargarMediaLegitimo ()**

Este método carga del archivo ./mediaLegitimo.dat la media de n-grams sospechosos encontrados en un entrenamiento anterior con tráfico legítimo.

- **cargarMediaAtaques ()**

Este método carga del archivo ./mediaAtaques.dat la media de n-grams sospechosos encontrados en el entrenamiento anterior con tráfico con ataques.

- **calcularMediaAtaques ()**

Calcula la media de n-grams sospechosos que surgen en el tráfico con ataques teniendo en cuenta la media calculada anteriormente. Este método es necesario porque el entrenamiento con ataques se hace de uno en uno.

2.2.5. Md5

El módulo Md5 se encarga de procesar el n-gram que se le introduce y devolver la función hash de su contenido. Es decir, realiza los cálculos necesarios para realizar el resumen del n-gram. Una vez calculado lo pasa al módulo Análisis que lo utilizará para actualizar el bloom filter o comparar el n-gram con el patrón correspondiente.

El único método del módulo se explica a continuación:

- **funcionHash (unsigned char*, const unsigned char* ngram)**

  Recibe un n-gram y devuelve el valor obtenido al calcular la función hash Md5 del contenido del n-gram.

2.2.6. Alertas

Este módulo se encarga de registrar en un archivo de texto la información relativa a las alertas generadas por el módulo Análisis. El archivo se llamará alertas.txt y contiene información sobre la hora y fecha de la alerta, y el número de n-grams sospechosos y totales.

El método de este módulo se explica a continuación:

- **registrar_alerta (int, int, const u_char*, int)**

  Este método va actualizando el archivo de alertas cada vez que recibe una nueva. Es el encargado de escribir los datos de los paquetes sospechosos en dicho archivo.

2.3. Diagramas de secuencia

2.3.1. Secuencia del entrenamiento con tráfico legítimo

Para entrenar al sistema con tráfico legítimo la interacción entre módulos es la mostrada en el diagrama 2.2.3.

En primer lugar, el módulo inicio lanza el Sniffer para recoger el tráfico. El Sniffer carga el bloom filter para que lo use el módulo de análisis. A continuación, procesa, analizar y pasa el paquete los datos recogidos al módulo de análisis.

El módulo de análisis consulta el bloom filter y manda al módulo md5 que haga la función hash para resumir el paquete.

Una vez resumido el contenido del paquete, el módulo análisis pasa la información al Sniffer que a su vez, guarda la media del legítimo.

Por último, se libera la memoria usada tanto por el sniffer como por el módulo Inicio.

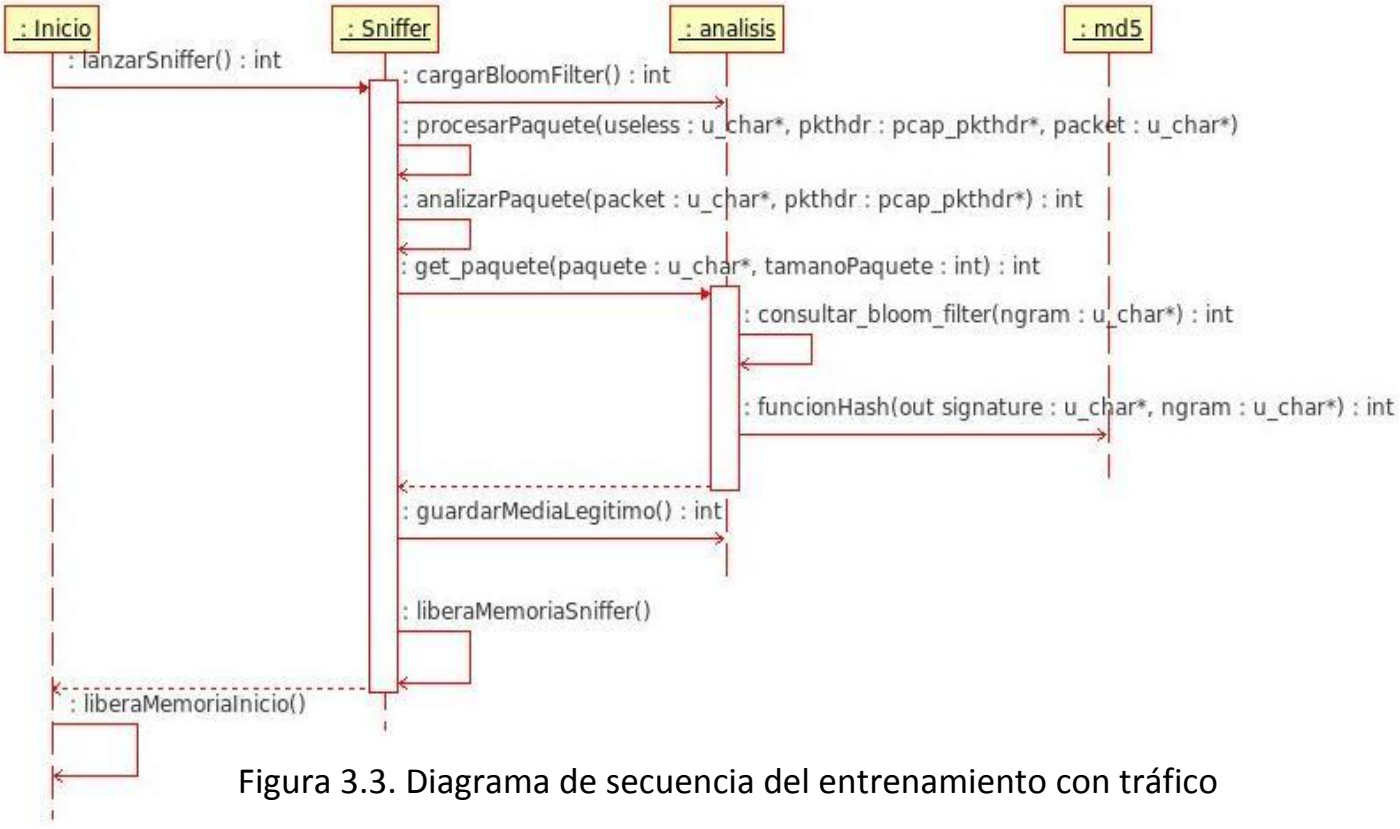

Figura 3.3. Diagrama de secuencia del entrenamiento con tráfico

2.3.2. Secuencia del entrenamiento con ataques

Para entrenar al sistema con tráfico con ataques la interacción entre módulos es la mostrada en el diagrama 2.2.4.

En primer lugar, el módulo Inicio lanza el Sniffer para recoger el tráfico. El Sniffer carga el bloom filter para que lo use el módulo de análisis. A continuación, procesa, analiza y pasa el paquete con los datos recogidos al módulo de análisis.

El módulo de análisis consulta el bloom filter y indica al módulo md5 que haga la función hash para resumir el paquete.

Una vez resumido el contenido del paquete, el módulo análisis pasa la información al Sniffer que a su vez, calcula la media de los ataques. El módulo Análisis actualiza la media de los ataques.

Por último, se libera la memoria usada tanto por el sniffer como por el módulo Inicio.

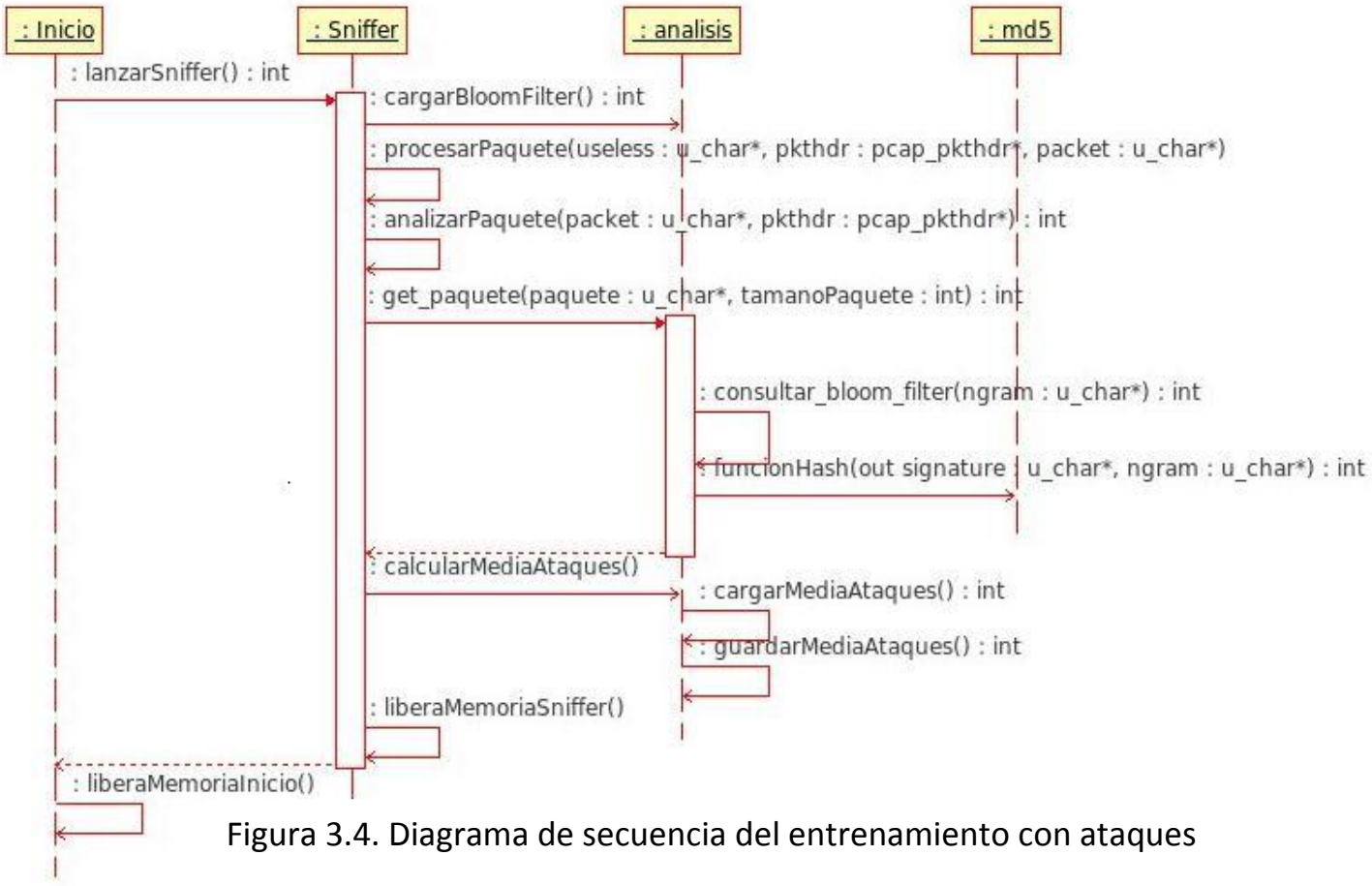

Figura 3.4. Diagrama de secuencia del entrenamiento con ataques

2.3.3. Secuencia del proceso de detección

Para el proceso de detección la interacción entre módulos es la mostrada en el diagrama 2.2.5.

En primer lugar, el módulo Inicio crea al archivo de Log, lanza Snort y lanza el Sniffer.

Una vez lanzado, el Sniffer carga el contenido del bloom filter y las medias del tráfico legítimo y de los ataques en el módulo Análisis. A continuación procesa y analiza los paquetes para pasar los datos obtenidos al módulo Análisis.

El módulo Análisis consulta el bloom filter y ordena al módulo MD5 el resumen del contenido del paquete, es decir, su función hash.

Una vez resumido el paquete y comparado con los datos del bloom filter, el módulo Análisis comprueba si hay que lanzar una alerta y si es así la lanza y la registra.

Por último, se libera la memoria usada tanto por el sniffer como por el módulo Inicio.

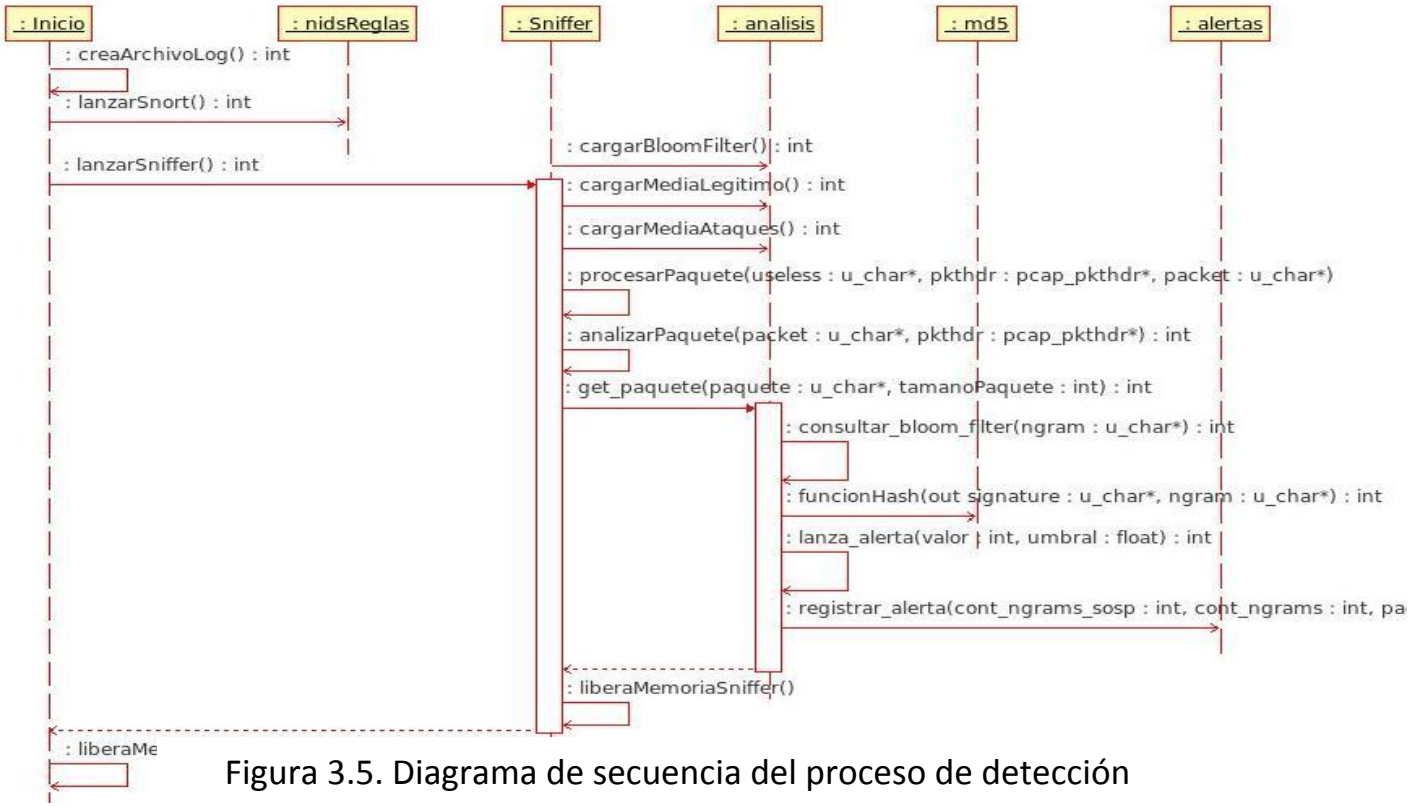

Figura 3.5. Diagrama de secuencia del proceso de detección

3. IMPLEMENTACIÓN

La implementación del prototipo se ha realizado en lenguaje C mediante el entorno de programación Eclipse para Linux, ya que el prototipo está pensado para funcionar en sistemas Unix/Linux. La elección de C como lenguaje se debe a que el rendimiento y la velocidad del sistema eran un requisito indispensable para poder analizar grandes cantidades de tráfico en tiempo real. C al ser un lenguaje de bajo nivel nos permite obtener un rendimiento suficiente para llevar a cabo este análisis.

A continuación explicamos los puntos más importantes y difíciles de la implementación.

3.1. Llamadas al sistema

Para capturar los paquetes que analizará el detector de anomalías hemos implementado un sniffer. Para conseguir capturar los paquetes (ya sea de la red o de un archivo pcap) hemos usado la librería libpcap que implementa una serie de llamadas al sistema que explicamos a continuación:

- **pcap_dump()**. Registra en un archivo pcap el paquete que se le pasa por parámetro.
- **pcap_open_live().** Abre el interfaz de red que se le pasa por parámetro para comenzar a capturar paquetes. Devuelve un descriptor que se usará para operar con este interfaz.
- **pcap_open_offline().** Abre el archivo que se indica por parámetro para empezar a capturar paquetes. Devuleve un descriptor que se usará para operar con ese archivo.
- **pcap_dump_open()** Abre el archivo que se le pasa por parámetro para registrar los paquetes leídos por el descriptor de archivo/interfaz que también se pasa por parámetro.
- **pcap_loop()**. Captura uno a uno los paquetes que se encuentran en el interfaz/archivo que se le pasa por parámetro. Para cada uno de estos paquetes se le aplica la función indicada en la llamada. Para

conseguir capturar todo el tráfico que pasa por la red introducimos -1 en la condición de parada del bucle con lo que conseguimos un bucle infinito.

3.2. Bloom Filter

El bloomFilter es una estructura de datos que creamos para modelar el tráfico legítimo de la red.

Conceptualmente se trata de una colección de bits los cuales se pueden acceder individualmente. Para poder implementarlo, utilizamos un array de caracteres, ya que C no contempla el tipo de datos bit o boolean y el tipo *char* es el que menos bits ocupa (8).

Sabiendo que cada carácter se compone de 8 bits, utilizamos un array de tamaño igual al de una octava parte del que realmente queremos y utilizamos las funciones a nivel de bit &, | y >> para poder acceder a cada bit del carácter individualmente. Esto lo logramos usando un índice exterior que selecciona un carácter del array y un índice interior que selecciona un bit dentro de un determinado carácter.

Además del bloomFilter principal que usamos como modelo de tráfico legítimo usamos una estructura de datos auxiliar a la que denominamos bloomFilterX. Esta estructura auxiliar se usa para llevar la cuenta del numero de n-grams de cada tipo aparecido durante el entrenamiento y se implementa con un array de enteros con las mismas posiciones que bits hay en el bloom filter, es decir con un tamaño 8 veces mayor que el del bloom filter original.

3.3. Función Hash MD5

Como hemos comentado anteriormente utilizamos el bloomFilter como modelo del tráfico legítimo y en él iremos anotando los n-grams que han aparecido durante el entrenamiento. Sin embargo, el número de posibles n-grams que pueden aparecer son demasiado y sería imposible

crear una estructura de datos que pudiera contenerlos todos (para n=7 las posibles combinaciones son de 2^{56}). Por ello es necesario una función hash que mapee cualquier posible combinación de un n-gram en una estructura más pequeña.

La función que utilizamos nosotros es MD5, que sabemos que es una función hash universal y que se puede usar sin problemas con claves grandes.

La implementación de esta función la hemos conseguido de la dirección: http://svn.freenode.net/hyperion/trunk/src/md5.c. Dicha implementación realiza el algoritmo explicado anteriormente usando los tipos de datos soportados por C.

3.4. Estadística

Para rellenar el bloom filter utilizamos medidas estadísticas que nos permiten despreciar ciertas apariciones remotas de algunos n-grams durante el entrenamiento. De esta forma conseguimos que nuestro sistema sea más robusto frente al ruido y, por tanto, el tráfico de entrenamiento no tenga que estar totalmente libre de ataques, aunque siga siendo muy recomendable.

También usamos métodos estadísticos para obtener el umbral de detección dinámicamente con los datos del entrenamiento.

Esta es la parte de nuestra propuesta que más se aleja de la propuesta de Anagram y la explicamos a continuación:

Durante la primera fase de entrenamiento se rellena el bloomFilterX con el número de apariciones de cada n-gram del tráfico analizado. Tras hacer esto hallamos la media de aparición de cada uno de los n-gram y la desviación típica respecto de esa media.

En una segunda pasada por el bloomFilterX eliminamos las apariciones que sean mayores que la suma de la media más la desviación típica y ponemos las posiciones correspondientes del bloomFilter a 1. Con esto

logramos eliminar aquellas apariciones de un determinado n-gram que sean demasiado altas y que podrían aumentar demasiado el valor de la desviación típica.

Una vez hecho esto se vuelve a calcular la media de los n-grams restantes del bloomFilterX así como su desviación típica. Con estos valores fijamos un umbral mínimo que será igual a la media menos la desviación y volvemos a procesar el bloomFilterX. En esta ocasión, sólo las posiciones con una frecuencia mayor que la del límite fijado serán puestas a 1 en el bloomFilter, eliminando aquellas apariciones poco significativas y que podrían deberse a ataques o a elementos aislados.

Para obtener el umbral de detección también usamos medidas estadísticas. En primer lugar obtenemos la media de los n-grams sospechosos en paquetes legítimos y la guardamos en un archivo. Esta representará una cota inferior del umbral de detección.

Por otro lado obtenemos la media de los n-grams sospechosos en el tráfico de entrenamiento con ataques. Para hacer esto, cogemos el número mayor de n-grams sospechosos que encontramos en un solo paquete en cada ataque (los ataques del entrenamiento están separados). De esta forma intentamos conseguir que solo salte una alerta por cada ataque encontrado.

Una vez calculados estos dos valores, es decir, una vez que conocemos la media de n-grams sospechosos en el tráfico legítimo y en el tráfico con ataques determinamos el umbral de detección. Para esto calculamos la media entre los dos valores anteriores. En la fase de detección un paquete se considerará sospechoso si tiene una cantidad de n-grams sospechosos superior a este valor.

V

PRUEBAS

1. DATASETS

Para el entrenamiento y la comprobación de los resultados de nuestro sistema hemos elaborado un dataset (conjunto de pcaps representativos del tráfico y de los ataques conocidos) con la herramienta de captura de tráfico Wireshark.

Wireshark es una herramienta que permite ver el tráfico que pasa a través de una red mediante una interfaz gráfica muy intuitiva. Además tiene la ventaja de ser software libre, disponer de infinidad de manuales de uso tanto en inglés como en español y un gran reconocimiento en el mundo de la auditoria de sistemas. Por todo esto decidimos usar esta aplicación para capturar el tráfico que formará parte de nuestros datasets.

Para realizar las distintas etapas del entrenamiento del sistema hemos necesitado crear varios conjuntos de datos: unos con tráfico limpio y otros con tráfico con código malicioso.

Para la creación de los datasets de tráfico limpio se capturó el tráfico de un ordenador doméstico durante varios días. Para poder crear un patrón de comportamiento fiable se realizaban todos los días acciones parecidas y visitas a las mismas páginas web. La acciones capturadas son las siguientes:

- Tráfico generado por aplicaciones de intercambio de archivos P2P.
- Envío de correo electrónico mediante Hotmail con ficheros adjuntos (.doc, .pdf, .mp3, .jpeg...).
- Lectura de correo electrónico en Hotmail con ficheros adjuntos (.doc, .pdf, .mp3, .jpeg...)
- Navegación por páginas web de todo tipo (as, marca, mundo deportivo, diario sport, elpais, el mundo, abc, larazon, fdi ucm, ucm, upm, uam, forocoches, todoexpertos, yahoorespuestas, comunidad de Madrid, gobierno de españa, the washington post, nba, universidad de Berkeley, wikipedia.es, wikipedia.org, cinetube, peliculasyonkis, seriesyonkis, vagos…)

- Descarga de programas y de aplicaciones del conocido servidor Softonic.

- Visualización de videos en youtube.

En total se construyen 3 datases de aproximadamente 300MB cada uno.

- Dataset1.
  Destinado al entrenamiento del Bloomfilter.

- Dataset2.
  Destinado a determinar el umbral de n-grams sospechosos en el tráfico limpio.

- Dataset3.
  Destinado para pruebas, es decir para analizarlo como si fuera tráfico normal y determinar el porcentaje de falsos positivos del sistema.

Para la captura del tráfico con ataques teníamos la necesidad de tener conjuntos de datos con un solo ataque aislado en cada uno para poder entrenar y probar bien el sistema. Debido a esto decidimos descargar algunos virus y programas peligrosos conocidos de las siguientes páginas web:

www.worst-viruses.2ya.com

www.rigacci.org/comp/virus

Estos programas estaban comprimidos por lo que para poder capturarlos subimos los virus descomprimidos al conocido *file hosting* MegaUpload y capturamos la descarga de estos creando un archivo pcap por cada virus.

El listado completo de virus capturados es el siguiente: melissa, nimda, loveletter, sasser.b, happy99, netsky.z, blaster, sobig, Bomberman, Bloodlust, DnDdos, MXZ, MXZ II, MMR, OwNeD, Zip Monsta, HDKP4, Die 3, Click 2, Gimp, BitchSlap, NetBus 2.0 Pro, NetDevil 1.5, NYB, Parity.b, WYX.b, Stoned.a, PingPong.a, Form.a, Parity.a, Bye, Junkie.1027,

HLLC.Crawen.8306, Macro.Word97.Thus.aa, Macro.Word97.Ethan, Macro.Word.Cap, Macro.Word97.Marker-based, Macro.Word97.Marker.r, Macro.Office.Triplicate.c, Trojan.Win32.VirtualRoot, JS.Fortnight.b, Trojan.PSW.Hooker.24.h, Trojan.JS.Seeker.o, JS.Trojan.Seeker-based, Trojan.Win32.DesktopPuzzle, JS.Trojan.Seeker.b, Backdoor.SdBot.aa, Backdoor.Zenmaster.102, Backdoor.DonaldDick.152, Backdoor.DonaldDick.15, Backdoor.Buttman, Worm.Win32.Fasong.a, Worm.Win32.Lovesan.a, Win32.Xorala, Win32.FunLove.4070, Worm.Win32.Muma.c, Win95.Dupator.1503, Win32.HLLP.Hantaner, Joke.Win32.Errore, Joke.Win32.Zappa, Worm.Bagle.Z, IIS-Worm.CodeRed.a, I-Worm.Moodown.b, I-Worm.NetSky.d, I-Worm.Rays, I-Worm.Sober.c.dat, I-Worm.LovGate.i, IIS-Worm.CodeRed.c, I-Worm.Sober.c, I-Worm.Mydoom.a, I-Worm.Mimail.a, I-Worm.Tanatos.dam, I-Worm.Swen, Worm.P2P.SdDrop.c, I-Worm.Sobig.f, I-Worm.Klez.e, I-Worm.Sobig.b, I-Worm.Tanatos.b, I-Worm.Fizzer, Worm.Win32.Opasoft.a.pac, I-Worm.Tettona, Worm.Bagle.AG, Trojan.ZipDoubleExt-1, Worm.Mytob.IV, orm.Mytob.BM-2, Worm.SomeFool.Q, Trojan.W32.PWS.Prostor.A, Worm.Mydoom.AS, Worm.Mytob.V.

Este conjunto de virus se puede clasificar en las siguientes categorías:

- Viruses.
  Malware que tiene por objeto alterar el normal funcionamiento de la computadora, sin el permiso o el conocimiento del usuario.

- Worms.
  Malware que tiene la propiedad de duplicarse a sí mismo.

- Nukers.
  Nombre genérico para ataques de tipo DoS en TCP/IP.

- Trojans.
  Malware capaz de alojarse en computadoras y permitir el acceso a usuarios externos, a través de una red local o de Internet, con el fin de recabar información o controlar remotamente a la máquina anfitriona.

- Macro viruses.
  Virus elaborados mediante un macro lenguaje, por ejemplo, archivos .doc con código malicioso.

Finalmente, una vez descargados todos los códigos maliciosos y capturados en archivos pcap se dividieron en dos conjuntos de datos: uno para el entrenamiento (determinar el umbral de n-grams sospechosos) y otro para realizar las pruebas de nuestro sistema y comprobar la eficiencia del mismo.

2. RESULTADOS

Después del proceso de entrenamiento, pruebas y diversas depuraciones obtuvimos los siguientes resultados: 2,4% de falsos positivos, 89,47% de código malicioso detectado.

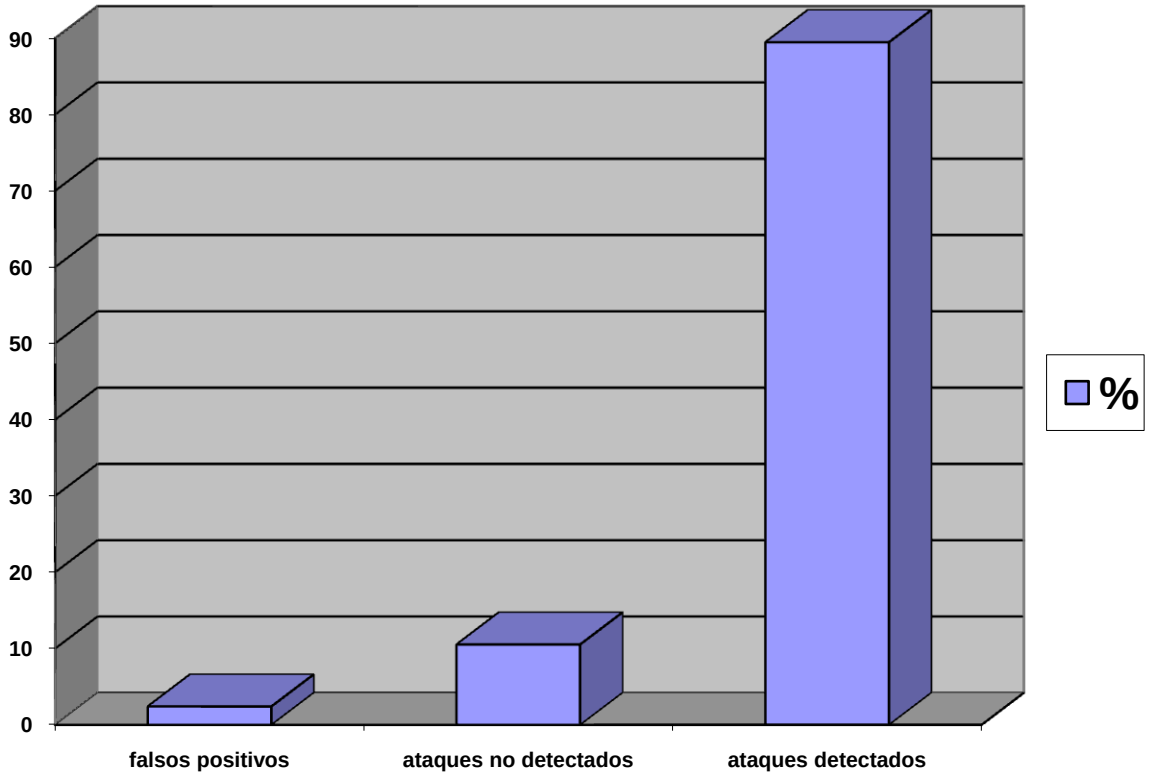

Figura 4.1. Resultado de las pruebas

En función de la clasificación de los ataques los resultados obtenidos serían los siguientes:

- Backdoors: Detectados 1, Total 2.
- Nuckers. Detectados 2, Total 3.
- Macro Viruses. Detectados 2, Total 3.
- Trojans. Detectados 4, Total 5.
- Viruses & Woorms. Detectados 21, Total 21.

En la figura 4.2 se pueden ver gráficamente estos resultados.

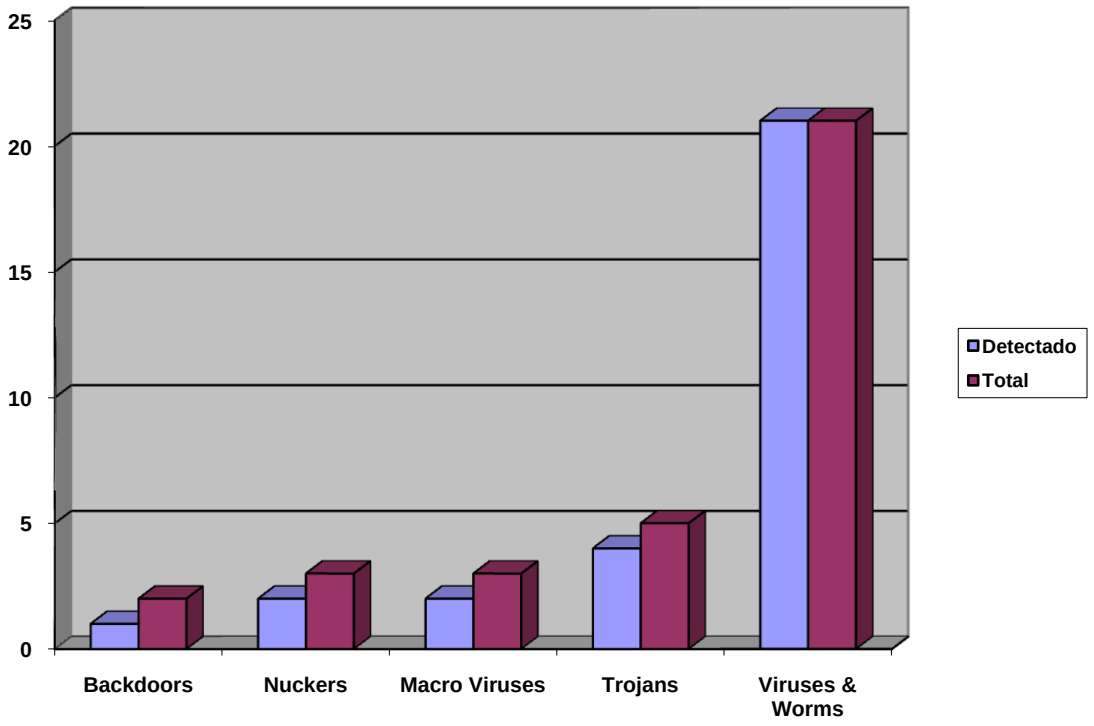

Figura 4.2. Resultados por tipos de ataques

VI

CONCLUSIONES

1. CONCLUSIONES

Una vez finalizado el proyecto llega el momento de exponer nuestras conclusiones sobre el trabajo realizado durante todo este año.

En general, pensamos que los objetivos marcados al principio se han cumplido de forma satisfactoria. La idea de crear un IDS híbrido que manejara las dos principales técnicas de detección nos pareció buena desde el principio y una vez llevada a la práctica los resultados muestran que así es.

Por otro lado, debemos destacar el enorme trabajo de documentación realizado. La temática del proyecto era totalmente desconocida para nosotros, sin embargo hemos conseguido aprender y plasmar en esta memoria gran cantidad de conocimiento sobre los IDS, las diferentes técnicas y Snort.

En definitiva, creemos que el trabajo realizado ha merecido la pena y que el prototipo desarrollado puede ser útil para la protección de cualquier red. Asimismo, admitimos que aún queda mucho trabajo por hacer en este campo y que el prototipo se puede mejorar para aumentar su eficiencia.

2. FUTUROS TRABAJOS

Siguiendo la línea de estudio de este proyecto se proponen mejoras y evoluciones del prototipo que permitan mejorar tanto su alcance como su rendimiento.

El primer objetivo que se debe perseguir a la hora de mejorar la propuesta es conseguir que la correlación entre las alarmas generadas por Snort y por el detector de anomalías se haga de forma automática y eficiente.

Una forma de conseguir esto sería la conversión del detector por anomalías en preprocesador de Snort. Esto permitiría una mayor correlación entre los eventos de los dos sistemas y asegurarnos más de la veracidad de las alertas. Además, el detector de anomalías podría beneficiarse del tratamiento de los paquetes que realizan otros preprocesadores de Snort.

Entre las mejoras del detector por anomalías estarían la posibilidad de realizar el entrenamiento de forma automática, es decir sin tener que introducir el tráfico de entrenamiento con ataques manualmente.

Otra dirección que puede tomarse para mejorar el sistema es la implementación de técnicas más específicas para diferentes tipos de ataques así como la generalización del prototipo para detectar otras intrusiones que no se basen en el código malicioso.

Por último, decir que la evolución del sistema híbrido debería llegar a combinar el NIDS creado con un HIDS que examine más a fondo el comportamiento de los terminales importantes del sistema informático. Una correlación correcta y rápida entre los dos tipos de IDS aumentaría en gran medida el rendimiento del sistema y la seguridad del sistema informático.

BIBLIOGRAFIA

- D. BOLZONI y S. ETALLE. *"Approaches in Anomaly-based Network Intrusion Detection Systems"*

- J.T. YAO, S.L. ZHAO y L.V. SAXTON *"A study on fuzzy intrusion detection*".
  Belur V. Dasarathy. Orlando, Florida, USA. Abril 2005

- D.E. DENNING. *"An intrusión-Detection Model".*
  IEEE Press. Piscataway, NJ, USA. 1987

- J. BEALE. *"Snort 2.1 Intrusion detection".*
  Syngress. USA. Mayo 2004

- Z. ZHUANG, Y. LUO, M. L y, CG WENG. *"A Resource Scheduling Strategy for Intrusion Detection on Multi-core Platform".*
  IEEE Computer Society. Washington D.C, USA. 2008

- Y.K. PENYA, P.G. BRINGAS. *"Integrating Network Misuse and Anomaly Prevention"*
  Deusto Technology Foundation. Bilbao, España. 2008

- ASHFAQ, ROBERT, MUMTAZ, QASIM, SAJJAD y KHAYAM. *"A Comparative Evaluation of Anomaly Detectors under PortScan Attacks".*
  NUST. Rawalpindi, Pakistan. 2008

- J.ZHANG, M.ZULKERNINE y A. HAQUE. "*Random-Forests-Based Network Intrusion Detection Systems*".
  TELUS, Toronto, ON M5B 1N9, Canada. Septiembre 2008

- K. WANG y S. J. STOLFO. "Anomalous Payload-based Network Intrusion Detection"
  Computer Science Department, Columbia University

- V.A. SKORMIN, D.H. SUMMERVILLE, y J. S. MORONSKI. *"Detecting Malicious Codes by the Presence of Their "Gene of Self-replication""*
  ECE Watson School, Binghamton University Binghamton, NY 13902

- Y. ZHANG, T. LI[1], J. SUN y R. QIN. *"An FSM-Based Approach for Malicious Code Detection Using the Self-Relocation Gene"*
  Sichuan University. China

- S. BEZOBRAZOV, V. GOLOVKO. *"Neural Networks for Artificial Immune Systems: LVQ for Detectors* Construction"
  Brest State Technical University, Bulgaria.

- M.F. MARHUSIN, D. CORNFORTH, H. LARKIN. *"Malicious Code Detection Architecture Inspired by Human Immune System"*
  School of ITEE. Canberra. Australia

- A. LÓPEZ MONGE. *"Aprendiendo a programar con Libpcap"*
  Deusto. Bilbao, España. 2005

- K. SCARFONE, P. MELL. *"Guide to Intrusion Detection and Prevention Systems (IDPS)"*

- C. SCOTT, P. WOLFE, B. HAYES. *"Snort for dummies"*
  Julio. 2004

- *SnortTMUsers Manual 2.8.3*
  The Snort Project, Septiembre, 2008

- *J.PEREZ AGUDÍN, C. MIGUEZ PEREZ y A. MATAS GARCÍA. "La biblia del Hacker"*
  Anaya. 2005

- SecurityFocus <www.securityfocus.com>

- Snort <www.snort.org>

- Wikipedia <www.wikipedia.org>

APENDICE I. INSTALACION DE SNORT 2.8.3.2

En este apéndice se explican paso a paso las acciones a realizar para instalar Snort 2.8.3.2 en un sistema Linux. A continuación se enumeran todas las acciones que hay que hacer y los comandos que hay que ejecutar para llevarlas a cabo:

1. Obtenemos privilegios de administrador

   sudo –i

2. Si durante el proceso de instalación nos da un error porque no tenemos un compilador, ejecutamos los siguientes comandos:

   aptitude update

   aptitude -y install build-essential

   aptitude -y install flex bison

   aptitude -y install libpcre3 libpcre3-dev

3. Descargamos todos los ficheros que vamos a necesitar:

   wget http://www.netfilter.org/projects/iptables/files/iptables-1.4.2.tar.bz2

   wget http://www.packetfactory.net/libnet/dist/deprecated/libnet-1.0.2a.tar.gz

   wget http://downloads.sourceforge.net/pcre/pcre-7.8.tar.gz?use_mirror=surfnet

   wget http://www.tcpdump.org/release/libpcap-1.0.0.tar.gz

   wget http://www.snort.org/dl/snort-2.8.3.2.tar.gz

4. Descomprimimos todos los archivos descargados en el punto 3:

   tar -xvf iptables-1.4.2.tar.bz2

   tar -zvxf libnet-1.0.2a.tar.gz

   tar -zxvf pcre-7.8.tar.gz

   tar -zxvf libpcap-1.0.0.tar.gz

   tar -zxvf snort-2.8.3.2.tar.gz

5. Borramos los archivos descargados una vez lo hemos descomprimido:

   rm -r iptables-1.4.2.tar.bz2

   rm -r libpcap-1.0.0.tar.gz

   rm -r libnet-1.0.2a.tar.gz

   rm -r pcre-7.8.tar.gz

   rm -r snort-2.8.3.2.tar.gz

6. Para cada uno de los ficheros descomprimidos anteriormente, salvo para el de Snort, ejecutamos los siguientes comandos desde su directorio:

   ./configure

   make

   make install

7. Nos metemos en la carpeta de iptables-1.4.2 y después en la carpeta de libipq y hacemos:

   make

   make install

8. Para el fichero de Snort ejecutamos:

   ./configure

   make

   make install

9. Descarga de las reglas de Snort

Una vez realizados los 8 pasos anteriores ya deberíamos tener Snort instalado en nuestro sistema. Sin embargo, para que funcione correctamente debemos descargarnos las reglas con las que analizará el tráfico. Sin los ficheros de reglas al intentar ejecutar el programa en modo NIDS o en modo Inline nos dará un error indicando que no puede encontrar dichos archivos. Para los modos de ejecución Sniffer Mode y Packet Logger Mode no tendremos ningún problema aunque no dispongamos del juego de reglas.

Para descargarnos el fichero con reglas de Snort podemos ir a la página oficial (www.snort.org), acceder a la sección “Rules”, y después a la sección “Download Rules”.

En este momento observaremos que las reglas se clasifican en 4 categorías:

1. Sourcefire VRT Certified Rules - The Official Snort Ruleset.

   Este conjunto de reglas es el oficial de Snort y está completamente actualizado. Sin embargo para descargarlo

necesitamos estar suscritos y pagar la correspondiente cuota anual.

2. Sourcefire VRT Certified Rules – The Oficcial Snort Ruleset (registred user release).

   Estas reglas únicamente pueden ser descargadas por miembros registrados. La diferencia entre ser miembro subscrito y ser miembro registrado es que los últimos pueden acceder a las actualizaciones de las reglas un mes después de que estas sean publicadas. Además solo se permite la descarga de un juego de reglas cada 24 horas. El proceso para registrarse es sencillo y rápido, como el de cualquier otra comunidad web.

3. Source VRT Certified Rules – The Official Snort Ruleset (unregistered user release).

   Estos conjuntos de reglas están muy desactualizados pero pueden descargarse sin necesidad de registrarse en la página web.

4. Community Rules.

   Son reglas desarrolladas por miembros de la comunidad de Snort. En principio es recomendable usar los juegos dados por los creadores.

Una vez que seleccionemos el conjunto de reglas a descargar, obtendremos un archivo con las reglas. Debemos descomprimir este archivo en la carpeta donde está instalado Snort.

Después de la descarga de reglas ya estamos preparados para probar nuestra instalación de Snort lanzando el programa en cualquiera de los 4 modos de ejecución que tiene.

APENDICE II: MANUAL DE USO

El programa diseñado es una aplicación que se ejecuta en la consola por línea de comandos al uso de la mayoría de las aplicaciones usadas en entornos Linux.

Para ejecutarlo se escribirá por consola el nombre de la aplicación seguido por las opciones de uso que queramos especificar. La sintaxis del comando que lanza la ejecución del programa es la siguiente:

NIDS [opciones]

A la hora de escribir el comando las opciones puede ir en cualquier orden. La lista de posibles opciones y su descripción se explican a continuación.

-h

ayuda: muestra por consola el manual de uso de la aplicación, es decir, las opciones permitidas y su significado. El uso de esta opción no puede combinarse con otras.

-m

modo: la opción –m deberá ir seguida por un espacio y alguno de los siguientes valores: entrenamiento, detección o pruebas. Esta opción es siempre obligatoria excepto cuando usamos –h. Explicamos a continuación cada uno de estos modos:

- Entrenamiento: con esta opción se ejecuta el programa en modo entrenamiento. El sistema debe estar entrenado antes de lanzar otros modos de ejecución. El modo de entrenamiento consta de varios submodos que se explicarán más adelante en la opción –s.
- Detección: con esta opción se ejecuta el programa en modo detección. Este modo captura paquetes de la red y los analiza para intentar encontrar comportamientos sospechosos. Para ejecutar

este modo de ejecución, el sistema debe estar entrenado. Consta de varios submodos que se explicarán en la opción –s.

- Pruebas: con esta opción se ejecuta el programa en modo de prueba. Este modo analiza un archivo especificado que debe contener tráfico previamente capturado. Se utiliza principalmente para probar el funcionamiento del sistema.

-s

submodo: la opción –s deberá ir seguida de un espacio y alguno de los siguientes valores: normal, legitimo, ataques o promiscuo. Esta opción se utiliza junto con los valores de modo entrenamiento y detección. Esta opción no es obligatoria a la hora de lanzar el programa (si no se especifica submodo se utilizará el normal). A continuación se explica el funcionamiento de cada submodo y el modo al que pertenece:

Submodos de entrenamiento:

- Normal: al usar este submodo hacemos que el archivo fuente, especificado mediante la opción –r (obligatoria en el modo de entrenamiento), se utilice para rellenar el bloomfilter.
- Legitimo: al usar este submodo hacemos que el archivo fuente, especificado mediante la opción –r, se utilice para obtener la media de los n-grams sospechosos aparecidos al procesar tráfico legítimo.
- Ataques: al usar este submodo hacemos que el archivo fuente, especificado mediante la opción –r, se utilice para obtener la media de los n-grams sospechosos aparecidos al procesar el tráfico con ataques. Es recomendable que cada vez que se ejecute el entrenamiento mediante esta opción el archivo fuente solo contenga un ataque. Debido a esto, se deberá ejecutar este modo de ejecución por cada ataque que tengamos para entrenar el sistema.

Submodos de detección:

- Normal: al usar este modo solo se procesarán los paquetes que se dirigen a la maquina en la que está instalado el IDS. El resto de paquetes que pasan por la red se ignorarán.
- Promiscuo: al usar este modo ponemos la tarjeta de red en modo promiscuo de forma que procesaremos todos los paquetes que circulen por la red no solo los dirigidos al equipo en el que está instalado el sistema.

-r

archivo fuente: la opción –r deberá ir seguida por un espacio y la ruta completa del archivo fuente donde estará guardado el tráfico a procesar. Esta opción es obligatoria en los modos de entrenamiento, el archivo contendrá el tráfico de entrenamiento, y de pruebas, el archivo contendrá los paquetes que se analizarán.

-l

archivo de logs: la opción –l deberá ir seguida por un espacio y por la ruta completa del directorio donde se crearán los archivos de log generados por el sistema. El usuario debe asegurarse que dicho directorio existe. Esta opción solo se puede utilizar en los modos de detección y de pruebas, y no es obligatoria. Si no se especifica ningún directorio los archivos se crearán en ./logs.

Las archivos que se crean en cada ejecución son: alert.txt (creado por Snort), *fecha*Alertas.txt y *fecha*Paquetes.txt (creados por el detector de anomalías) donde *fecha* será la fecha dia-mes-año en que se ejecutará la detección.

El contenido de estos archivos se explica a continuación:

- alert.txt. Contendrá las alertas generadas por Snort y una descripción del tipo de alerta.

- fechaAlertas.txt. Contendrá las alertas generadas por el detector de anomalías. La información que se guarda es la hora y la puntuación del paquete sospechoso (cuanto más alto sea este valor más probabilidades habrá de que se trate de un verdadero ataque).
- fechaPaquetes.txt: En este archivo se registrarán los paquetes que han producido una alerta en el detector de anomalías junto con la hora en la que se ha producido la alerta.

-i

interfaz de red: la opción –i deberá ir seguida por un espacio y el nombre del interfaz de red que se usará para capturar los paquetes. Esta opción solo se usa con el modo de detección y no es obligatoria (si no se especifica se tomará por defecto eth0). Nuestro sistema solo funcionará con interfaces de red Ethernet los interfaces inalámbricos wlan no son válidos para esta aplicación.

Ejemplos

Se indican a continuación distintas llamadas a la aplicación que se muestran como ejemplo:

- Para ejecutar el sistema en modo entrenamiento con submodo normal y rellenar el bloom filter con los paquetes del archivo /home/usuario/legitimo.pcap usaremos el siguiente comando:

./NIDS –m enrenamiento –r /home/usuario/legitimo.pcap

- Para ejecutar el sistema en modo entrenamiento con submodo legítimo y calcular la media de los n-grams sospechosos, que usaremos como cota inferior del umbral de detección, con los datos del archivo /home/usuario/legitimo2.pcap usaremos el siguiente comando:

./NIDS –m entrenamiento –s legitimo –r
/home/usuario/legitimo2.pcap

- Para ejecutar el sistema en modo entrenamiento con submodo ataques y calcular la media de los n-grams, sospechosos que usaremos como cota superior del umbral de detección, con los datos del archivo /home/usuario/legitimo2.pcap usaremos el siguiente comando:

./NIDS –m entrenamiento –s ataques –r /home/usuario/ataques2.pcap

- Para ejecutar el sistema en modo pruebas y coger el tráfico del archivo /home/usuario/pruebas.pcap usaremos el siguiente comando:

./NIDS –m pruebas –r /home/usuario/pruebas.pcap

- Para ejecutar el sistema en modo detección con submodo normal cogiendo los paquetes del interfaz eth0 y registrando las alarmas en el directorio ./logs usaremos el siguiente comando:

./NIDS –m detección

- Para ejecutar el sistema en modo detección con submodo promiscuo cogiendo los paquetes del interfaz eth0 y registrando las alarmas en el directorio /home/logs usaremos el siguiente comando:

./NIDS –m detección –s promiscuo –i eth0 –l /home/logs