Escuela Superior de Ingeniería

Segundo Ciclo en Ingeniería Informática

Detección de Amenazas de Seguridad mediante Procesamiento de Eventos Complejos

Curso 2011-2012

Jose Antonio Dorado Cerón

Cádiz, 29 de febrero de 2012

Escuela Superior de Ingeniería

Segundo Ciclo en Ingeniería Informática

Detección de Amenazas de Seguridad mediante Procesamiento de Eventos Complejos

Departamento: Lenguajes y Sistemas Informáticos.
Director del proyecto: Juan Boubeta Puig.
Autor del Proyecto: Jose Antonio Dorado Cerón.

Cádiz, 29 de febrero de 2012

Fdo.: Jose Antonio Dorado Cerón

Índice general

Índice de figuras

Índice de cuadros

*

Capítulo 1

Introducción

1.1. Contexto y Motivación

Actualmente el campo de la seguridad informática tiene una gran importancia, que aumenta día a día a medida que avanzan las tecnologías. Prácticamente todos los equipos informáticos ya sean destinados a empresas o para usuarios domésticos se encuentran conectados a la red, que cada día nos ofrece nuevas posibilidades para trabajar, comunicarnos o tener acceso a cualquier servicio y recurso.

Con todo esto, cada día aumentan las amenazas que se encuentran en la red, haciendo de la seguridad informática un campo esencial e indispensable, ya que el robo de datos, destrucción de equipos o cualquier otro ataque, pueden causar un daño irreparable y en muchos casos suponer un gran desastre para la víctima.

Para tomar conciencia de la verdadera importancia que tiene este campo, tan sólo es necesario ver algunos informes sobre las pérdidas que se han producido por culpa de ataques informáticos. En 2010 según un estudio realizado por *Symantec* [36], tan sólo en España se generaron unas pérdidas de 5.900 millones de euros sumando el coste directo y el tiempo empleado en la recuperación de los sistemas. En ese mismo año, un estudio realizado por *Norton [36]* indica que en todo el mundo, las pérdidas directas por

ataques informáticos ascienden a 114.000 millones de dólares, sin tener en cuenta el tiempo de recuperación de los sistemas ni las pérdidas generadas en las empresas por no poder operar durante ese tiempo, lo que aumentaría exponencialmente esa cifra.

Con estos datos podemos hacernos una idea de la auténtica importancia que tiene poseer un buen sistema de seguridad. Podemos definir este área como la parte de la informática que mediante una serie de herramientas, reglas, protocolos y estándares se dedica a minimizar los riesgos en los sistemas computacionales, todo lo relacionado con éstos y con toda la información que contienen.

Los objetivos de la seguridad informática son tres, por una parte proteger la infraestructura computacional, entendiendo por esto, velar por el correcto funcionamiento de los equipos informáticos y toda su estructura. En este punto no sólo nos referimos a proteger dichos sistemas de posibles ataques, sino de cualquier tipo de desastre o fallo. Otro objetivo de la seguridad informática es proteger a los usuarios que utilizan estos sistemas. Y el objetivo, quizás el más importante, es el referente a la protección de la información que contienen los equipos informáticos, bases de datos... ya que en algunos casos, esas pérdidas pueden llegar a ser irreparables.

En cuanto a las amenazas a las que está sometido un sistema informático, podemos enumerar las siguientes: el propio usuario que hace uso del equipo, cualquier tipo de desastre natural o ajeno a la informática, programas maliciosos en forma de virus, troyanos, gusanos... la manipulación de usuarios mediante ataques de ingeniería social y los ataques a través de la red. En este proyecto nos vamos a centrar en estos últimos, y vamos a trabajar con los tipos de ataques más importantes y usuales que se suelen realizar a través de la red, ya sea del tipo de una amenaza externa, donde el atacante es ajeno a la empresa, o mediante una amenaza interna, que en otros casos podrían ser mucho más difíciles de ser detectados debido al conocimiento de la red por parte del atacante, sus niveles y permisos de acceso y la dificultad de ser detectados estos ataques mediante mecanismos como *firewall* al realizarse dentro de la misma red.

Una vez especificado el tema que vamos a tratar y cuál es el problema que vamos a abordar tenemos que pensar con qué herramientas y tecnologías vamos a contar. Este punto es clave, una mala elección aquí puede condenar una buena idea al fracaso. En el campo de la seguridad un factor determinante es el tiempo, y es que de nada nos vale detectar y alertar de una amenaza una vez que el atacante ha logrado su objetivo, por lo tanto a la hora de realizar este proyecto tenemos que priorizar en todo momento el tiempo de respuesta, el cual debemos minimizar lo máximo posible para actuar en tiempo real.

Para poder llevar a cabo esta propuesta vamos a tratar de explicar la idea de las principales tecnologías que vamos a utilizar. En primer lugar, tenemos que tener en cuenta que el sistema en el que operemos puede ser atacado y que debemos tener una herramienta que detecte dichos ataques y a su vez nos sirva como fuente para trabajar a partir de la información que nos reporte. Para ello vamos a utilizar una herramienta de monitorización del tráfico de red, concretamente *Snort* [21], que es un sistema de detección de intrusos o *Intrusion Detection System* (IDS) que se distribuye bajo licencia GPL [8] y está basado en regla. Esta aplicación va a registrar en tiempo real cualquier intento de intrusión o ataque a nuestro sistema para posteriormente generar una salida que utilizaremos con el objetivo de detectar los posibles ataques y actuar en consecuencia. Para más información referente a esta herramienta véase el capítulo 2. Otra de las principales motivaciones a la hora de desarrollar este proyecto es la utilización de tecnologías emergentes, que sean potentes y tengan un futuro importante por delante. Por ello hemos decidido utilizar el procesamiento de eventos complejos o *Complex Event Processing* (CEP) **[51]**, que nos permitirá trabajar en tiempo real con la información que obtengamos de *Snort*. Con esta tecnología vamos a poder obtener información de diferentes fuentes, definir una serie de patrones en *Event Processing Language* (EPL) que es el lenguaje de programación que nos ofrece el motor de CEP para poder diseñar patrones de eventos y, a partir de ellos, detectar situaciones relevantes de nuestro sistema en tiempo real.

En concreto, en este proyecto se enviarán al motor CEP las alertas generadas por *Snort* (en forma de eventos) y mediante esta tecnología analizaremos y correlacionaremos estos eventos para detectar situaciones relevantes (eventos complejos) que concuerden con algunos de los patrones de eventos definidos

previamente. Finalmente, a partir de los eventos complejos detectados actuaremos en consecuencia. A la hora de desarrollar este proyecto podemos pensar en un arquitectura orientada a servicios o *Service Oriented Architecture* (SOA) como solución, pero como ha sido demostrado [63], el tiempo real es un concepto clave y por lo tanto vamos a utilizar CEP . La solución para integrar SOA con CEP es utilizar un bus de servicio empresarial o *Enterprise Service Bus* (ESB) que nos va a permitir combinar ambos enfoques. Un ESB es una arquitectura software que podemos usar para integrar y comunicar diferentes aplicaciones, tecnologías y enfoques. En nuestro proyecto vamos a trabajar con Mule ESB [13] que nos va a permitir trabajar en tiempo real e integrar las diferentes tecnologías con las que vamos a trabajar. Los eventos que se produzcan en el IDS *Snort* serán enviados al ESB Mule donde se tratarán y se enviarán al motor CEP Esper [5, 45] para la detección de los eventos complejos y posteriormente notificar en tiempo real al responsable de la seguridad del sistema de la alarma que se ha detectado en referencia al ataque producido.

1.2. Objetivos

Este proyecto se enmarca dentro de algunas de las líneas de investigación del "Grupo UCASE de Ingeniería del Software" (TIC-025) [30] de la Universidad de Cádiz, en concreto, las arquitecturas orientadas a servicios y las arquitecturas dirigidas por eventos. El objetivo principal de este proyecto es la creación de un sistema de seguridad capaz de detectar los ataques de redes más comunes. Para poder alcanzar este objetivo con éxito debemos cumplir los subobjetivos que se enumeran a continuación:

1. Desarrollar un sistema multiplataforma que sea capaz de actuar en tiempo real. Esto es, debe ser capaz tanto de detectar cualquier tipo de alerta en el momento en el que se produzca como actuar de la manera más rápida posible en el caso de que detectemos un ataque a nuestro sistema.

2. Estudiar y analizar cuáles son los principales ataques que se realizan a través de la red.

3. Diseñar patrones de eventos mediante el lenguaje de programación EPL para los principales ataques a los que puede estar sometido nuestro sistema, como pueden ser ataques DOS, DDOS, *Smurf*, *Flood*, FTP...

4. Crear una arquitectura que integre CEP y SOA mediante el uso de un ESB.

5. Integrar de manera efectiva *Snort* con el ESB.

1.3. Alcance

Este proyecto se crea como una herramienta destinada tanto a usuarios particulares como a empresas, con el objetivo de aumentar la seguridad de sus sistemas minimizando el riesgo ante una serie de ataques que se pueden producir a través de la red y teniendo la capacidad de responder de manera rápida y efectiva.

La seguridad informática es una de las áreas más grandes que existen y no es nuestro objetivo abarcar todas sus ramas, sino centrarnos en los ataques a través de la red. Por lo tanto la utilización de esta herramienta no es sustitutiva de otras destinadas a la protección de los equipos, ya que no es nuestra intención el desarrollo de un sistema similar a un *firewall* ni ningún antivirus. En este proyecto vamos a realizar una herramienta que nos permita una rápida respuesta ante la posibilidad de ataques.

Actualmente se ha creado este software para la detección de ocho de los ataques más usuales en este entorno, concretamente podemos detectar los siguientes ataques: denegación de servicio, ataque distribuido de denegación de servicio, ataques *smurf*, ataque *land*, ataque *supernuke*, escaneo de puertos TCP, ataque *flood* al email y ataque al puerto FTP (véase sección 5.5.5), aunque está diseñado de manera que sea escalable, es decir, que podamos diseñar nuevos patrones de eventos y aumentar así de manera sencilla el número de ataques a detectar.

Este sistema es multiplataforma, por lo que es posible ejecutarlo en cualquier sistema operativo y aumentar así su nivel de seguridad.

1.3.1. Licencias

A lo largo de este proyecto se han utilizado diferentes plataformas y tecnologías, siendo libre el uso de todas ellas. El proyecto ha sido desarrollado en un sistema operativo de tipo Linux, más concretamente Ubuntu que se distribuye bajo una licencia de tipo GPL.El software utilizado para monitorizar el tráfico de red, *Snort*, también se distribuye bajo licencia GPL. Por otro lado, Mule ESB también posee una licencia de software libre, en este caso, CPAL [3].

Para el procesamiento de eventos complejos hemos utilizado el motor de Esper, el cual se distribuye bajo licencia GPL 2.0, y todas las bibliotecas de Java utilizadas también poseen licencia de software libre.

1.4. Visión general

Una vez vista la introducción al proyecto y las tecnologías de las que se hacen uso, vamos en primer lugar a tratar el estado del arte en el área que nos ocupa dicho proyecto. Posteriormente trataremos en profundidad las nuevas tecnologías a las que hacemos referencia.

En el siguiente capítulo vamos a presentar el calendario que se ha seguido para el desarrollo del proyecto seguido de una descripción general de éste.

A continuación trataremos el proceso de análisis, diseño, implementación y pruebas que se han seguido.

Para finalizar incluiremos un resumen de los puntos más importantes del proyecto, las conclusiones, las líneas futuras a seguir y, por último, los manuales de instalación, de usuario y del desarrollador.

1.5. Glosario

1.5.1. Acrónimos

BSD Berkeley Software Distribution

CEP Complex Event Processing

CPAL Common Public Attribution License

DDOS Distributed Denial Of Service

DOS Denial Of Service

EDA Event Driven Architecture

EJB Enterprise Java Beans

EPL Event Processing Languaje

ESB Enterprise Service Bus

FTP File Transfer Protocol

GPL GNU General Public License

GUI Graphical User Interface

HIDS Hosts Intrusion Detection System

HTML Hyper Text Markup Language

HTTPS Hyper Text Transfer Protocol Secure

ICMP Internet Control Message Protocol

IDE Integrated Development Enviroment

IDS Intrusion Detection System

IMAP Internet Message Access Protocol

IP Internet Protocol

JBI Java Business Integration

JMS Java Message Service

LAN Local Area Network

LAND Local Area Network Denial

MAC Media Access Control

NAC Network Access Control

NIDS Network Intrusion Detection System

OSGi Open Services Gateway initiative

OSI Open System Interconnection

PING Packet Internet Groper

POJO Plain Old Java Object

POP Post Office Protocol

REST Representational State Transfer

SCDL Service Component Definition Language

SMPT Simple Mail Protocol Transfer

SOA Service Oriented Architecture

SOAP Simple Object Access Protocol

SQL Structured Query language

SSH Secure SHell

SSL Secure Sockets Layer

TCP Transmission Control Protocol

UDP User Datagram Protocol

WSDL Web Services Description Language

XML eXtensible Markup Language

1.5.2. Definiciones

Broadcast Es un modo de transmisión de información en el que el emisor envía la información de manera simultanea a varios nodos sin que esta información tenga que transmitirse de uno a otro.

Estándar Serie de normas que se establecen para garantizar el acoplamiento de diferentes elementos que se construyen de manera independiente.

Estándar Serie de normas que se establecen para garantizar el acoplamiento de diferentes elementos que se construyen de manera independiente.

Evento Es un suceso en el sistema, ya sea un mensaje que se envía o una interacción con un usuario. También puede ser considerado algo que se espera que ocurra pero no llega a suceder.

Evento complejo Un evento que es una abstracción de otros eventos, es decir, es el resultado de la suma de varios eventos simples correlacionados entre sí

Evento simple Un evento que no está compuesto de otros eventos.

Flujo de eventos Secuencia de eventos ordenados que pueden ser de un mismo tipo o no.

Patrón de evento Condiciones que han de cumplirse para detectar un evento complejo a partir de eventos simples.

Protocolo Conjunto de reglas que se usan para controlar las comunicaciones a través de la red.

Sniffer Software que registra la actividad que sucede en un ordenador o una red.

Spoofing Es la utilización de técnicas de suplantación de la identidad con fines maliciosos.

*

Capítulo 2

Estado del arte

2.1. Sistemas de seguridad actuales

Actualmente el campo de la seguridad informática está muy extendido, existiendo numerosos estudios publicados y multitud de software destinado a la protección de los equipos informáticos. En cuanto a los sistemas de seguridad actuales, podemos dividirlos en dos ramas, los referentes a la seguridad física, que consisten en la aplicación de barreras físicas y la implantación de procedimientos de control para prevenir desastres como incendios, robos, terremotos... No vamos a profundizar en este tipo ya que nuestro proyecto no hace referencia a ella.

El otro tipo de seguridad a la que nos referimos es la seguridad lógica, en este caso los ataques van contra la información almacenada en nuestro equipo. Hay multitud de herramientas en este campo, vamos a ver cuál es la utilidad de cada una de ellas y así podremos enmarcar nuestro proyecto.

Uno de los tipos de herramientas más importantes son las de control de acceso, cuyo objetivo es unificar las diferentes tecnologías de seguridad como antivirus, *firewalls*, sistemas de autenticación con los usuarios reforzando así la seguridad de la red. Se trata de un conjunto de protocolos para asegurar los nodos de la red antes de que éstos accedan a dicha red, no permitiendo el acceso de alguno de éstos si no cumplen con las normativas de seguridad.

Gracias a estas políticas, normas y comprobaciones de seguridad sobre usuarios y recursos, se puede tener un control mucho mayor sobre los usuarios y dispositivos que acceden a la red y que pueden hacer éstos en ella.

Si bien estos sistemas de control de acceso a la red o *Network Access Control* (NAC) previenen la intrusión de posibles atacantes a nuestra máquina y minimiza el riesgo de contaminación de otros equipos mediante sistemas de replicación como gusanos informáticos, no exime de tener un buen antivirus y *firewall* instalados en el sistema, ya que no cumplen la misma misión. Dentro de estos sistemas NAC podemos distinguir los llamados de pre-admisión en los cuáles se inspeccionan las estaciones finales antes de que se les permita el acceso a la red, por ejemplo previniendo que un equipo que no tenga los sistemas de seguridad actualizados pueda conectarse a la red. El otro tipo son los sistemas de post-admisión los cuales trabajan según las acciones que realicen los usuarios una vez que se les ha proporcionado el acceso a la red. Con estos sistemas NAC se permite a la red que tome decisiones de control basadas en la inteligencia artificial. Los sistemas NAC más utilizados libres son *OpenNac* [19], *Packetfence* [20], *HUPnet [10]* y *NetReg* [15].

Otros sistemas de seguridad informática actuales son los relativos a la autenticación, en este caso sobre todo los referentes a la red. Estos sistemas lo que realizan es un proceso para confirmar que la persona, recurso o máquina que desea acceder a la red o a otro recurso es realmente quien dice ser. En la red, la autenticación se considera en tres pasos que son autenticación, autorización y contabilización o *Authentication, Authorization and Accounting* (AAA). En primer lugar, la autenticación por el cual se verifica la identidad digital de la persona, máquina o software que solicita el acceso. En segundo lugar, la autorización que proporciona el sistema para acceder a dichos recursos una vez comprobada la identidad del solicitante y, en tercer lugar, la auditoría que es la encargada de registrar todos los pasos y recursos a los que se accede. Hay que diferenciar el proceso de autenticación con el de autorización, el primero se refiere a comprobar la identidad de la persona o máquina, mientras que el segundo comprueba que dicha persona o máquina tiene los permisos suficientes para acceder a donde desea.

En cuanto a los sistemas de autenticación, actualmente hay varios tipos, por una parte están los referentes a una característica del usuario, normalmente estos sistemas se basan en la biometría que es la aplicación de técnicas matemáticas y estadísticas sobre los rasgos físicos o de conducta de un individuo. Otros tipos de autenticación son los referentes a algo que posee el usuario, como pueden ser tarjetas inteligentes, llaves electrónicas , *smartcards*... Y el tipo de autenticación más corriente es el referente a algo que se conoce como suele ser una contraseña.

Un sistema de seguridad que se utiliza bastante actualmente son los denominados sistemas de criptografía. Básicamente estos sistemas lo que hacen es transformar un texto mediante una serie de operaciones aritméticas de manera que el resultado no sea comprensible para un posible atacante que capture dicha información. En la red uno de los sistemas de criptografía que más se suelen utilizar es la firma digital. Esta firma digital se basa en la aplicación de un algoritmo matemático conocido como función *hash*. En la red, para enviar un paquete y asegurarnos de que el emisor es quien dice ser, éste puede firmarlo con su clave privada y su clave pública, siendo sólo necesario el conocimiento de la clave pública del emisor por parte del receptor para poder descifrar el mensaje y comprobar que la persona que lo envía es quien dice ser.

En el mercado actual, otro tipo de sistemas de seguridad informática son los *antispyware*, éstos tienen como objetivo detectar y eliminar los *spyware*, que son programas que tienen como función recopilar toda la información posible de la máquina que se infecta y enviar dicha información a un tercero sin consentimiento de la máquina atacada. Al igual que todos los sistemas mencionados con anterioridad, no se recomienda el uso exclusivo de estos sistemas, ya que como estamos viendo no tienen el mismo objetivo que los demás. Actualmente los *antispyware* más demandados en el mercado son *Spybot* [22], *Ad-Aware [1]* o *SpywareBlaster* [23].

Uno de los tipos de sistemas de seguridad que se utilizan en la actualidad y que tiene una gran importancia son los cortafuegos o *firewalls*, estos sistemas tienen por objetivo bloquear todos los accesos no autorizados a la red y poseen un conjunto de reglas en los cuales el administrador puede definir cuáles

son las comunicaciones que sí están autorizadas. Los cortafuegos examinan todo el tráfico que se produce en la red y bloquea aquello que no cumple con los criterios de seguridad. Hay diferentes clases dentro estos sistemas, algunos trabajan específicamente para ciertas aplicaciones específicas como por ejemplo para FTP. Otros son los que aplican ciertos mecanismos de seguridad cuando se establece alguna conexión TCP o UDP con el objetivo de que estas comunicaciones se realicen de manera más segura. Otros cortafuegos son los que funcionan al nivel de la capa 3 del modelo OSI o a la capa 2 del modelo TCP/IP, es decir, a nivel de red realizando filtros según los diferentes campos de los paquetes IP. También pueden trabajar a nivel de la capa de transporte haciendo uso de los puertos de origen o destino, o en el nivel de enlace de datos realizando en este caso filtros utilizando la dirección MAC. Otro tipo de cortafuegos son los que trabajan a nivel de aplicación teniendo filtros personalizados según sea el caso. En este caso suelen ser *proxys* que sirven para ocultar la dirección de red.

Si bien este tipo de software va a analizar el tráfico entrante en nuestro equipo, la utilización del proyecto que hemos desarrollado no está destinado a eliminar el uso de estos, ya que nuestro objetivo no es cortar las comunicaciones que establezcamos como peligrosas, sino monitorizar el tráfico de red y detectar así posibles ataques a nuestro sistema. Un cortafuegos puede prevenir alguna intrusión o limitar el acceso a la red pero no va a conocer que es lo que está pasando internamente en la red. En cambio nuestro sistema va a analizar estas comunicaciones de red para actuar en caso de comprobar que se está produciendo un ataque.

Posiblemente el tipo de sistemas de seguridad más extendidos en la actualidad son los antivirus, si bien no están tan relacionados con nuestro proyecto como los sistemas que hemos visto anteriormente, pero su mención es obligada debido a la importancia de su uso. Un antivirus es un software destinado para prevenir el contagio y propagación de virus informáticos, además de ser capaces en algunos casos de eliminar y desinfectar áreas del sistemas que hayan sido contagiadas. Su función se puede dividir en tres partes, en primer lugar la vacuna, que es un programa instalado y que actúa como filtro para todos los programas que se instalan o ejecutan. Otra de sus partes es el detector, que es el encargado de examinar los archivos con el objetivo de encontrar software malicioso en alguno de ellos. Por último encontramos el

eliminador, que es el encargado de destruir este tipo de software. En ningún caso la utilización de nuestro sistema de detección sería sustitutivo de un antivirus ya que las áreas en las que trabaja cada uno dentro de la seguridad informática son bastante distantes.

El último de los tipos de software destinados a la seguridad informática que podemos reconocer en la actualidad son los sistemas de detección de intrusos. Éstos tienen como objetivo detectar accesos a nuestro sistemas o red sin que hayan sido autorizados. Estos IDS poseen algún tipo de sensor, como puede ser un *sniffer* para poder obtener los datos sobre el tráfico de red. El funcionamiento de esta herramienta es el análisis de todo el tráfico de red el cual se compara con ataques conocidos para detectar así las acciones maliciosas. En muchos casos un IDS trabaja a la par que un *firewall*, creando así una potente herramienta de seguridad ya que se une la capacidad de bloquear que tienen los cortafuegos con la inteligencia artificial que poseen los IDS.

En la figura 2.1 podemos ver una arquitectura donde se encuentra tanto un cortafuegos como un NIDS para proteger nuestros equipos de posibles accesos no permitidos desde el exterior. Podemos diferenciar dos tipos de IDS, los sistemas de detección de intrusos basados en hosts o *Host-Based Intrusion Detection System* (HIDS) que funcionan dependiendo de los rastros que dejen los atacantes y los sistemas de detección de intrusos en red o *Network Intrusion Detection System* (NIDS) que están basados en red y son en los que nos hemos basado a la hora de desarrollar nuestro proyecto. En nuestro caso vamos a hacer uso de *Snort*, uno de los detectores de intrusos más conocidos y que también puede ser reconocido como un sniffer. Vamos a utilizar su función de analizar todos los paquetes que circulan por nuestra red y a través de la información que nos va a proporcionar, vamos a crear un sistema que sea capaz de detectar ataques mediante el procesamiento de eventos complejos. En la sección 2.2 vamos a profundizar en *Snort* y en otros sistemas para rastrear el tráfico de red.

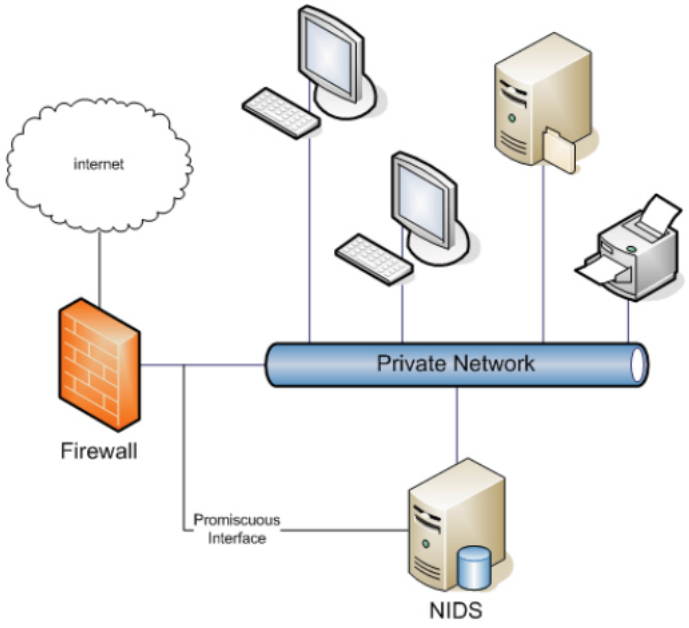


Figura 2.1: Arquitectura con cortafuegos y NIDS

Como hemos observado, si bien la cantidad de amenazas que existen actualmente son innumerables, también hay una gran cantidad de software en el mercado con la capacidad de minimizar el riesgo. Es totalmente recomendable el uso combinado de varios de estos sistemas que hemos visto. En nuestro caso nos vamos a centrar en el último punto para crear un sistema de detección basado en red a partir de un sniffer.

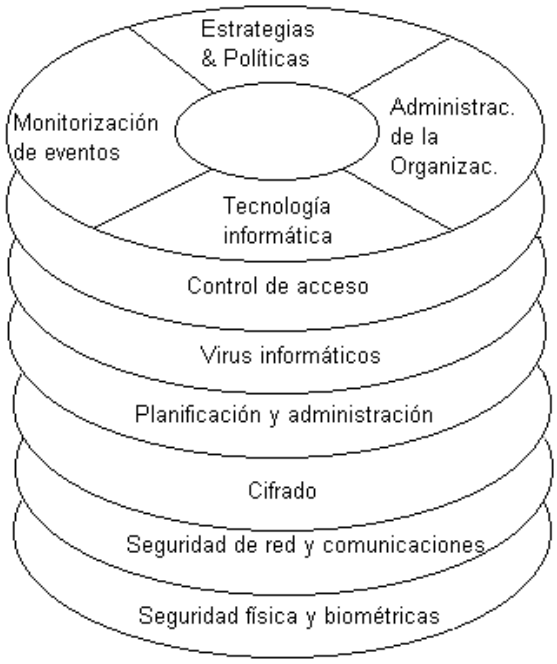


Figura 2.2: Estructura de seguridad

Hemos visto que la seguridad informática puede dividirse en muchas capas y que tenemos gran cantidad de software para cada una de ellas, en la figura 2.2 vemos la estructura de la que hemos hablado y cuáles son todos los puntos que debemos asegurar para tener un sistema totalmente protegido. Nosotros trabajaremos en las capas de más alto nivel, en la monitorización de eventos y control de acceso a nuestro sistema.

2.2. Snort

Como hemos indicado anteriormente, para el desarrollo de este proyecto vamos a utilizar una herramienta que monitorice el tráfico de red, concretamente *Snort*, y nos basaremos en las alertas que este programa genere para detectar los eventos a partir de ahí. Vamos a explicar en profundidad qué es *Snort*, como trabaja, cuáles son las alternativas actuales que se encuentran en el mercado y porqué lo hemos escogido. Posteriormente en veremos como se puede configurar y ejecutar para trabajar correctamente 9.

Snort es un *sniffer* o IDS que tiene como objetivo monitorizar todos los eventos que se produzcan en la red de un sistema informático y que puedan comprometer la seguridad de éste. Un IDS en general tiene una serie de patrones definidos en su motor, y lo que hace es ir comparándolos con los eventos que van surgiendo en el sistema, buscando alguno que se ajuste a los que tiene registrado y declarando así una actividad sospechosa. El tipo de seguridad que ofrece este tipo de sistemas es de prevención, ya que genera alertas sobre la seguridad del sistema.

En este proyecto vamos a ir mas allá, y si bien un software como *Snort* nos va a generar ciertos ficheros de alertas, nosotros vamos a crear una estructura SOA 2.0 en la que utilizando esta información como fuente, vamos a detectar eventos complejos y a realizar una acción como resultado de ello.

Como vimos en la sección 2, los IDS se encuadran en dos tipos, HIDS y NIDS, siendo *Snort* de la segunda clase. La funcionalidad de este tipo de sistemas y por consiguiente de *Snort* es la de capturar todos los paquetes que se transmiten en una red y analizarlos para encontrar patrones que se ajusten a cualquier tipo de amenaza. La forma de actuar es la de un dispositivo de red en modo promiscuo viendo todos los paquetes que circulen y trabajando no sólo a nivel de TCP/IP sino también a nivel de aplicación.

Estos IDS pueden ser clasificados en dos tipos según sea su respuesta, pasivos que son los que van a actuar enviando algún tipo de alerta y activos que son los que actúan directamente sobre el ataque, cerrando la conexión por ejemplo. *Snort* puede ser configurado para que actúe de cualquiera de las maneras, lo que le da bastante potencia y lo desmarca de muchos de sus competidores.

Un IDS por norma general tiene una arquitectura que podríamos dividir en cinco puntos. En primer lugar una fuente que se dedica a recoger los datos, en este caso un dispositivo de red. El segundo punto sería una serie de reglas donde figuren los patrones que pueden ser detectados como maliciosos. El tercer punto son los filtros para comparar los datos que se rastrean con los patrones que se encuentran almacenados en el sistema. El cuarto punto son los detectores de eventos anormales en el tráfico de red y por último, en el

quinto punto, tenemos los dispositivos que tienen la capacidad de generar las alarmas o actuar respecto al ataque.

Concretamente *Snort* podemos enmarcarlo en una arquitectura de tres niveles, donde en el primero se detectan y decodifican los paquetes. En el segundo nivel se encuentra el motor de detección donde se realizan las comparaciones con los patrones y en el tercer nivel se producen las respuestas de *Snort* en forma de alertas.

Otro punto interesante sobre estos IDS y sobre el sistema que hemos desarrollado es el lugar donde se debe colocar. En caso de estar colocado antes de un cortafuegos se detectaría todo el tráfico entrante y podría generar falsos positivos, en cambio si es colocado en una máquina posterior a este cortafuegos se minimizaría este riesgo. En caso de usuarios particulares y en un número importante de empresas, normalmente ambos están en la misma máquina. Para evitar las falsas alarmas, veremos posteriormente como hemos implementado algún mecanismo de este tipo en nuestra arquitectura.

Aunque posteriormente vamos a tratar las alternativas que nos ofrece el mercado, si nos centramos en *Snort* podemos decir que este sistema posee licencia GPL y puede funcionar bajo plataformas Windows y UNIX/Linux. Actualmente cuenta con una gran cantidad de filtros o reglas, muchas de ellas gratuitas. Podemos utilizar *Snort* tanto como NIDS o bien como un *sniffer* que capture los paquetes y guarde los ficheros de monitorización para su análisis. Una vez que *Snort* se encuentre instalado y en ejecución (ver anexo manual de instalación de *Snort* 9) podemos indicarle mediante diferentes opciones como queremos que nos muestre las alertas cada vez que capture alguna.

En cuanto a las alertas, como hemos comentado *Snort* funciona mediante reglas, ya sean paquetes completos que podemos descargar o bien reglas que hayamos desarrollado nosotros mismos. Podemos configurar *Snort* para que nos muestre la alerta en el formato que escojamos bien por pantalla o bien almacenándolos en la estructura de ficheros que se ha creado. En las alertas se almacena toda la información referente al riesgo detectado, como por ejemplo el tiempo, la clasificación de la alerta, las direcciones IP de origen y destino, los puertos... En la figura 2.3 podemos ver un ejemplo de una alerta lanzada

```
[**] [116:59:1] (snort_decoder): Tcp Window Scale Option found
with length > 14 [**]
[Priority: 3]
08/09-11:35:01.706839 192.168.1.33:61958 -> 192.168.1.35:1
TCP TTL:37 TOS:0x0 ID:60884 IpLen:20 DgmLen:60
**U*P**F Seq: 0x7EDA46B0 Ack: 0x9EC73F92 Win: 0xFFFF TcpLen: 40
UrgPtr: 0x0
TCP Options (5) => WS: 15 NOP MSS: 265 TS: 4294967296 0 SackOK
```

Figura 2.3: Ejemplo de alerta de *Snort*

por *Snort*. En nuestro proyecto cuando analicemos estos ficheros de alertas, tendremos que filtrar toda la información que no nos resulta importante para quedarnos con la que utilizaremos para detectar los eventos complejos.

Un tema muy importante de *Snort* y que lo diferencia de su competencia es la posibilidad de desarrollar tus propias reglas. *Snort* posee un lenguaje muy potente y flexible y que basándonos en una serie de normas nos permite que programemos algunas reglas que no se encuentren disponibles en el mercado. Para crear una regla tenemos que tener en cuenta por un lado la cabecera y por otro las opciones. La cabecera va a contener la acción de la regla, los protocolos que se utilicen, direcciones IP, máscaras de red, los puertos de origen y destino y la dirección de la operación. En cuanto a la sección de opciones, ahí tendremos que indicar los mensajes y toda la información necesaria para la toma de decisiones por parte de la alerta.

Vamos a mostrar un ejemplo de una regla de *Snort*, concretamente de una regla que controla el acceso al demonio de administración *mountd* de Unix, a través del cual un atacante podría obtener privilegios de administración en caso de un desbordamiento de memoria. La regla es la siguiente:

```
alert tcp any any -> 192.168.1.0/24 111 //
(content:"|00 01 86 a5|"; msg: "mountd access";)
```

Si la analizamos podemos diferenciar las siguientes partes. En la primera mitad observamos la cabecera, la acción de la regla es `alert`, el protocolo es

`tcp`, la dirección IP de origen y el puerto de origen pueden ser cualquiera. La dirección IP de destino es `192.168.1.0/24` y el puerto de destino `111`. La dirección de la operación es de entrada. En la segunda mitad de la regla podemos ver las opciones. *Content* nos permite buscar contenidos dentro del campo datos del paquete IP y la regla finaliza con el mensaje *msg*.

Hemos visto todo lo que nos puede aportar *Snort* y la potencia que tiene. En el anexo referente al manual de uso de *Snort* se puede ver en profundidad como se instala y se configuran todas las opciones. Ahora vamos a analizar cuales son las diferentes alternativas de uso a *Snort*.

Antes de entrar en el uso de otras herramientas del mismo carácter que *Snort*, tenemos que comentar una alternativa que valoramos implementar en el proyecto antes de decantarnos por el uso de *Snort*. Se pensó que como fuente de los datos para nuestro proyecto se podría utilizar *Syslog* [26], que es un protocolo de red que se usa en sistemas tipo UNIX para enviar mensajes de registro sobre una red IP. También hay posibilidades para sistemas tipo Windows al respecto como pueden ser *Syslserve [27]* o *WinSyslog* [33].

Las alertas que registra *Syslog* son referentes a la seguridad como por ejemplo intentos de acceso al sistema, fallos que se producen en él, errores, o cualquier información relativa a lo que ocurre en el sistema operativo. La estructura de los mensajes que genera *Syslog* es la siguiente, la prioridad que indica quien ha generado el mensaje y cuál es su importancia. La cabecera que indica el tiempo y el nombre de la máquina o dirección IP que genera el mensaje. El último campo es el texto donde se encuentra la información sobre el proceso que ha generado la alerta y posteriormente el contenido de dicha alerta.

Syslog posee varias características interesantes, como la variedad de mensajes que genera y su sencillez con lo que facilitaría su posterior trato. Pero hemos decidido que es mejor utilizar una herramienta de monitorización del tráfico de red por varias razones. *Syslog* tiene algunos fallos de seguridad y además si trabajásemos con este tipo de ficheros el tiempo de respuesta aumentaría bastante con lo cual ya no estaríamos trabajando en tiempo real, lo cual ha sido el motivo determinante para rechazar su uso. Otro problema que posee *Syslog* es el formato de las alertas que genera, en caso de que usáramos

esta herramienta dependeríamos de una versión concreta, ya que en caso de que se produjese una actualización de este sistema sería posible que el formato en el que se generasen las alertas cambiase y en ese caso nuestro sistema no podría procesarlas de manera adecuada. Otro de los motivos es la cantidad de información y la gran potencia que tienen las herramientas para la monitorización del tráfico de red, ya que son capaces de generar multitud de alertas y muchos escenarios diferentes. Por lo tanto, si bien es una opción interesante hemos decidido utilizar otro tipo de herramientas como sensores para nuestro proyecto. Ahora vamos a analizar otras herramientas de monitorización que se encuentran actualmente en el mercado y veremos porqué hemos escogido *Snort* de entre todas ellas.

A la hora de desarrollar el proyecto ya hemos visto que necesitamos un sensor o productor de eventos, y que éste va a ser una herramienta que sea capaz de analizar el tráfico de red y proporcionarnos la información necesaria para la detección de ataques mediante el procesamiento de eventos complejos. En el mercado actual hay muchas herramientas que pueden hacer esta función y una de las más importantes es *Tcpdump* **[28]**, que es un software capaz de capturar los paquetes en tiempo real y mostrarlos al usuario. En los sistemas tipo UNIX hace uso de la biblioteca *libpcap [12]* para la captura de paquetes, mientras que en sistemas tipo Windows, la herramienta hermana a *Tcpdump* seria *WinDump [31]* y la biblioteca de la que hace uso es *Winpcap* [32]. Esta herramienta proporciona la posibilidad a los usuarios de crear filtros para obtener una salida mas depurada. Gracias a esta herramienta podemos tanto depurar nuestra red y las aplicaciones que hacen uso de ellas como obtener los paquetes que circulan por la red y obtener datos. Es interesante reseñar que en sistemas tipo UNIX sólo el *root* puede hacer uso de esta herramienta.

Otra herramienta importante es *Darkstat [4]* que además de analizar el tráfico de la red, es capaz de generar un informe estadístico en HTML. Gracias a esto se pueden ver cuales son las direcciones IP que generan más tráfico y los puertos más utilizados. Gracias a este programa podemos obtener gráficos y resúmenes en diferentes ventanas temporales de los paquetes que se analizan. En cuanto a herramientas que proporcionen gráficos, *Traffic-Vis [29]* es un demonio que tiene la capacidad de monitorizar el tráfico TCP/IP y convertir dicha información en gráficos de diferentes formatos. También puede comprobar cuales son los *hosts* que se comunican y con que cantidad de datos lo

hacen.

Ngrep [16] es otra de las herramientas más usuales en este campo, lo más importante de este software es que es capaz de aplicar las características del *grep [9]* de GNU a la capa de red del modelo OSI. Esta herramienta permite el uso de expresiones regulares y que concuerden con el cuerpo de los paquetes. Una herramienta interesante al respecto es *Nwatch [18]* que es un analizador de puertos pasivo, aunque en este caso sólo es capaz de observar el tráfico IP y mostrar los resultados en función de los puertos. Este software ofrece alguna característica sobre los escáners de puertos habituales como puede ser que es capaz de coger los puertos que se abren solamente para la transmisión de datos. La forma en la que *Nwatch* funciona es permanecer activo mientras que no reciba una señal que le indique lo contrario y durante ese tiempo va siguiendo cada conexión de IP *host*/puerto que va descubriendo dentro de la interfaz por defecto. También puede mostrar un análisis de los patrones de uso para la supervisión de la red.

Una de las herramientas más conocidas en el mercado actual es *Wireshark* [34], éste es un analizador de protocolos de red multiplataforma y es capaz de capturar directamente los datos de una red y también puede obtener la información a partir de un archivo capturado previamente. Basa su potencia en que es capaz de soportar más de 400 protocolos y cuenta con una de las comunidades de usuarios más grandes en este tipo de software. Usa un sistema diferente para la visualización de los datos y para realizar las capturas de los paquetes en la red. También cuenta con una interfaz gráfica que hace que su uso sea mas amigable. Esta herramienta es comúnmente utilizada en conjunción con *Tcpdump*, ya que se puede aumentar la potencia utilizando dicha aplicación para capturar los paquetes desde la interfaz de red y posteriormente *Wireshark* para analizarlos.

Otras herramientas que merecen ser mencionadas en este apartado son *Ettercap [6]* que está especialmente diseñado para trabajar en redes LAN con switchs y tiene diferentes posibilidades a la hora de analizar el tráfico de red. En este caso es compatible con SSH para las conexiones seguras y con HTTPS para interceptar comunicaciones mediante SSL. Para trabajar con redes inalámbricas también tenemos una herramienta diseñada para ellas,

Kismet [11] que trabaja en Linux y es capaz de identificar redes de modo pasivo, recoger los paquetes e incluso detectar redes ocultas gracias al tráfico de datos.

Una vez que hemos visto la gran cantidad de herramientas a las que podemos acceder en el mercado es hora de decidir cual de ellas va a ser la base a partir de la que va a funcionar nuestro sistema. La opción escogida ha sido *Snort* por varias razones, es una herramienta como hemos visto en su análisis muy potente, que funciona bajo cualquier plataforma y eso es una característica muy importante para nuestro proyecto. Gracias a esta herramienta podemos trabajar en tiempo real sin lo cual no tendría sentido el desarrollo de nuestro software. *Snort* a diferencia de los demás que hemos visto nos proporciona una cantidad de reglas prácticamente infinitas ya que somos nosotros mismos los que podemos desarrollarlas y adaptarlas a nuestro caso de interés particular, aparte de que de manera gratuita nos proporcionan paquetes con gran cantidad de reglas.

La comunidad de usuarios que utilizan este producto es importante y creciente, además *Snort* recibe actualizaciones constantes lo que nos permite que nuestro sistema pueda ser seguro y no se quede obsoleto. Uno de los puntos positivos que nos ofrece *Snort* es su capacidad para ser configurado, como se puede ver en el anexo correspondiente podemos configurarlo para trabajar sólo con un paquete de reglas que queramos, podemos modificar la salida o hacer que trabaje de la manera que nosotros queramos en cada momento. El último de los puntos que debe cumplir la herramienta que escojamos es la salida que produce y *Snort* cumple a la perfección con nuestras necesidades, ya que proporciona ficheros de alerta completos y con los que no resulta complejo trabajar. Como veremos en los siguientes capítulos, hemos integrado *Snort* con un ESB consiguiendo así utilizar un nuevo enfoque que vamos a definir en la sección 2.3 en la cual *Snort* va a ser nuestro productor de eventos y a partir de ahí seremos capaces de detectar ataques mediante el procesamiento de eventos complejos.

2.3. SOA 2.0

Últimamente las arquitecturas orientadas a servicio se están utilizando cada vez más, ya que garantizan una buena solución cuando es importante la comunicación entre diferentes servicios o aplicaciones. Con esta arquitectura se pueden crear sistemas en los que los servicios pueden ser reutilizados y compartidos obteniendo así una buena implantación de la lógica de negocio en un entorno poco acoplado.

El objetivo principal de SOA es permitir la comunicación entre diferentes tecnologías existentes y futuras arquitecturas además de automatizar el procesamiento y ofrecer información a los usuarios cuándo necesiten interactuar con un proceso de negocio. Es decir, la misión de SOA es crear una arquitectura donde van a existir unos servicios que publiquen cierta información y otros servicios que sean capaces de consumir esa información.

Como hemos visto, en las arquitecturas SOA tenemos tres entidades importantes y bien diferenciadas. Por una parte tenemos el proveedor de servicios que son las aplicaciones que ofrecen un servicio. También existen los registros de servicios que son los encargados de publicar una descripción de los servicios que se ofrecen en un registro de servicios y por último están los consumidores de servicios que son las aplicaciones software que solicitan la ejecución de un servicio.

Para que estos servicios puedan comunicarse entre sí se debe ofrecer una definición fácil de comprender y una interfaz estándar que se encuentre totalmente abstraída de la implementación interna del servicio. Gracias a esto se pueden comunicar diferentes servicios con distintas tecnologías de manera rápida y sencilla, siendo independiente de la plataforma en la que se ejecuten.

Actualmente se tiende a pensar que estas arquitecturas sólo están pensadas para trabajar con servicios web, utilizando normalmente estándares como XML, SOAP, REST o WSDL [48], pero realmente no es necesario que se trabaje con este tipo de servicios, sino que se puede implementar una aplicación SOA utilizando cualquier tecnología orientada a servicios.

Gracias a este tipo de arquitecturas se pueden obtener muchos beneficios como puede ser la facilidad de integración de diferentes tecnologías, el poder crear aplicaciones de negocios basados en servicios de terceros, mejora en los tiempos para realizar cambios en los procesos y la capacidad de poder realizar cambios en la capa aplicativa de SOA sin alterar el proceso de negocio.

Si bien hemos visto todas las bondades que nos ofrecen las arquitecturas SOA, ésta no se adecua correctamente a nuestro proyecto debido a que dichas arquitecturas no están preparadas para sistemas en los que debemos estar analizando continuamente toda la información que fluye y trabajar prácticamente en tiempo real reconociendo cuáles son las situaciones críticas.

Este tipo de limitaciones se solucionan en otra arquitectura conocida como arquitectura dirigida por eventos o *Event Driven Architecture* (EDA), que definen una metodología para desarrollar aplicaciones en las que se deben transmitir eventos entre los componentes de servicios débilmente acoplados. La estructura general de esta arquitectura consta de un productor de eventos que publican en un gestor que recibe los eventos y los envía a un consumidor de eventos. Además el gestor puede almacenar los eventos que no se consuman debido a la falta de disponibilidad del consumidor e intentarlo más tarde.

Como podemos ver en la tabla 2.1 hay bastantes diferencias entre arquitecturas SOA y arquitecturas EDA. Algunas de las más importantes y que debemos conocer es que por ejemplo EDA trabaja con eventos en tiempo real, en SOA tanto los clientes como los servidores están débilmente acoplados a diferencia de EDA que se encuentran totalmente desacoplados. En SOA no se garantiza la entrega de la información mientras que en EDA sí. Además en las arquitecturas SOA una vez que el flujo comienza ya no se permiten entradas imprevistas mientras que las arquitecturas EDA son mucho más flexibles ya que pueden reaccionar a nuevas entradas externas aunque el proceso se encuentre activo.

Cuadro 2.1: Diferencias entre SOA y EDA

SOA	EDA
Basada en servicios demandados, normalmente síncronos.	Basada en eventos en tiempo real, normalmente asíncronos e impredecibles.
Clientes y servidores débilmente acoplados.	Clientes y servidores desacoplados.
Petición/respuesta uno-a-uno (*pull*).	Publicación/suscripción muchos-a-muchos (*push*).
Bidireccional, pero no garantiza la entrega.	Unidireccional, entrega de evento garantizada.
Ideal para procesamiento secuencial y aplicaciones compuestas en las que están disponibles todos los sistemas asociados y servicios.	Ideal para procesos paralelos, eventos (disparadores y uniones) y manejo de excepciones entre sistemas desconectados.
Usa metadatos de interfaces.	Usa metadatos de descriptores de eventos.
El cliente dirige flujos.	El receptor determina el flujo.
Cerrado a una entrada imprevista una vez que el flujo ha comenzado.	Puede reaccionar a nuevas entradas externas mientras el proceso esté activo.

Gracias al uso de CEP podemos aprovechar las características que nos ofrece esta tecnología que nos permite procesar y analizar una gran cantidad de eventos. Además podemos hacer esto en tiempo real detectando las situaciones críticas, que en este caso serían los posibles ataques al sistema. Dicho esto podemos pensar que la solución no se encuentra en utilizar una u otra, sino en integrar ambas arquitecturas, lo que se conoce como SOA 2.0.

Nuestro trabajo ahora se debe centrar en conjuntar SOA 2.0 con CEP, y para ello tenemos un elemento que debe ser la pieza angular sobre la que se apoye dicha integración, un ESB. En la figura 2.4 basada en [63] podemos ver una estructura en la que existen unos productores de eventos que pueden ser aplicaciones, servicios web o sensores y que a partir de unos protocolos que

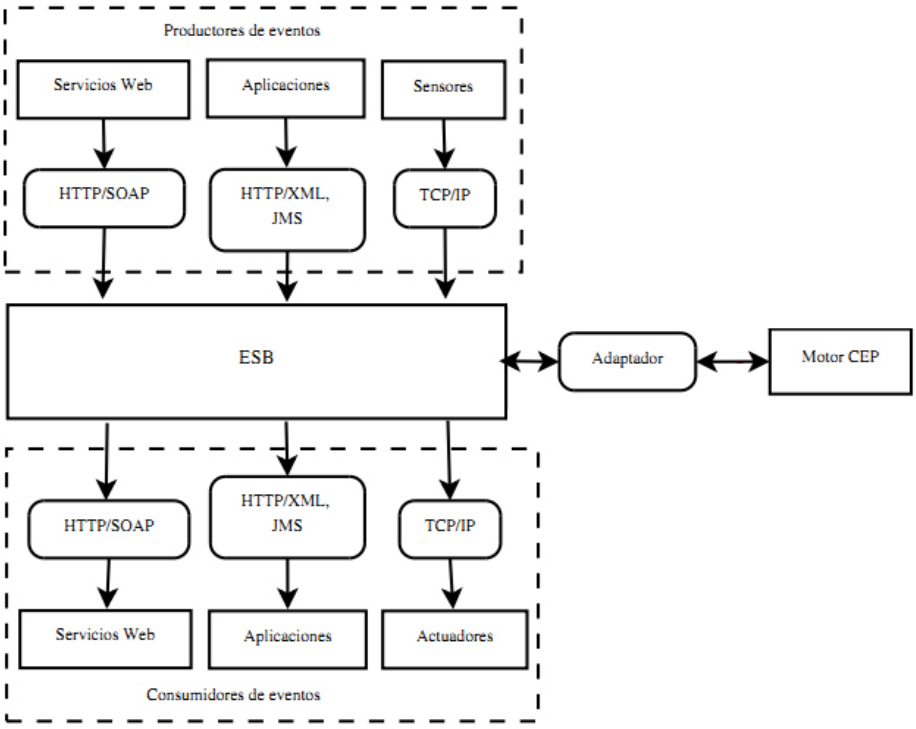


Figura 2.4: Arquitectura SOA 2.0 y CEP

entienda el ESB van a enviarle la información. En esta etapa se pueden transformar estos mensajes que se reciben en eventos para que posteriormente sean procesados por CEP. También se pueden priorizar los eventos enriqueciendo los mensajes y evitando así algunos problemas que pueden surgir como los cuellos de botella. Y por último el ESB va a ser el encargado de enviar estos eventos a todos los que se hayan suscrito para recibir la información.

En nuestro caso, vamos a tener un productor de eventos que sería *Snort*, el cual publica su información en forma de alertas, y nosotros la enviamos al ESB donde la trataremos y transformamos a eventos, que son enviados al motor de CEP para su procesamiento y detección de los eventos complejos. Posteriormente publicaremos esa información para actuar de manera adecuada.

2.4. ESB

Como hemos visto en el capítulo anterior, vamos a necesitar un ESB para poder realizar la integración de CEP con SOA 2.0 Un ESB permite la combinación de diferentes arquitecturas y proporciona una capa de abstracción sobre la implementación del sistema de mensajes de la lógica de negocio.

No se puede decir que un ESB implemente una arquitectura como SOA o EDA, sino que proporciona las técnicas necesarias para su implementación o para la integración de algunas de ellas. Podríamos decir que el ESB es la capa intermedia que consigue comunicar las diferentes aplicaciones entre ellas. Su estructura es la que podemos ver en la figura 2.4 donde el ESB recibe un mensaje a través de un proveedor de información que puede ser una aplicación, un servicio web o un sensor. En el ESB se trataría estos mensajes transformándolos para posteriormente enviarlos mediante un manejador de eventos a un consumidor que esté suscrito a nuestro ESB.

Actualmente hay diversos autores que discrepan sobre lo que debe ser considerado un ESB, existen una corriente que dice que debe ser un producto software a partir del cual se permita crear la arquitectura del sistema que se va a diseñar, en cambio hay otros que indican que el ESB simplemente es un estilo de arquitectura. Aun así hay algunas características que siempre deben poseer los ESB como son:

- Deben aportar mecanismos para soportar los protocolos de transporte, ya sean síncronos o asíncronos.
- Deben tener capacidad de encaminamiento, es decir, ser capaces de encaminar según sean los contenidos de los mensajes, las normas o las políticas.
- Un punto importante es su capacidad de transmitir mensajes e incluso de poder enriquecerlos con nuevo contenido.
- Deben de poder implementar los procesos complejos de la lógica de negocio.

- Tienen que incluir herramientas capaces de transformar protocolos, adaptadores o mapeadores de servicios.
- Deben ser capaces de trabajar con eventos.
- Se tiene que poder trabajar con diferentes servicios que sean presentados como un único servicio.
- Tienen que ofrecer servicios que aporten seguridad al sistema como pueden ser el cifrado, las entregas confiables o la administración de transacciones.
- Deben tener capacidad de administración, monitorización, auditoría o registro.

El uso de un ESB como hemos visto es indispensable a la hora de desarrollar un proyecto con esta arquitectura, pero el hacer uso de ello incluye algunas ventajas y desventajas [46]. En la parte positiva podemos incluir que gracias al uso de estas herramientas se pueden integrar aplicaciones y servicios ya existentes de manera rápida y mas barata que sin el uso de ellas. Provee a los sistemas de escalabilidad, es decir, se pueden incrementar sus funciones en cualquier momento sin afectar al rendimiento. Está basado en estándares y suelen incluir algunas herramientas predefinidas que ayudan a realizar la implementación del producto final. Además tienen una gran capacidad de configuración lo que hace que podamos ajustarlo a nuestras necesidades.

En la parte negativa del uso de estos ESB debemos indicar que no es adecuado para pequeñas estructuras o modelos simples, además tenemos que administrar de manera correcta los servicios que van a proveer los mensajes. Son necesarios conocimientos adicionales para poder configurar y trabajar con un ESB y a la hora de obtener el producto final son un elemento más que puede producir fallos . Sin embargo, en la actualidad están emergiendo herramientas gráficas, como *MuleStudio* [14], que permiten configurar y trabajar con los ESB de una forma fácil, amigable e intuitiva.

Visto esto, antes de comenzar a desarrollar nuestro producto tenemos que plantear la cuestión de en qué escenarios es necesario la utilización de un ESB. Podemos decir que su uso es recomendado en tres tipos de situaciones:

ESB	Descripción
Apache ServiceMix	Implementación de Apache JBI con componentes JBI
Apache Synapse	ESB orientado a servicios web basados en Apache Axis2
Apache Tuscany	ESB implementado mediante especificaciones SCA
ChainBuilder ESB	ESB basado en JBI con herramientas gráficas para el desarrollo
FUSE ESB	ESB de IONA basado en Apache ServiceMix
JBoss ESB	Implementación de un ESB basado en mensajes de JBoss
Mule	ESB ligero con un modelo de implementación a medida
Open Adapter	Provee diferentes adaptadores para integrar soluciones
Open ESB	Imlpementación JBI de Sun para trabajar con NetBeans
PEtALS	ESB basado en JBI de OW2
Spring Integration	ESB para trabajar con el framework Spring
WSO_2 ESB	ESB basdado en Apache Synapse

Cuadro 2.3: ESB de código abierto en el mercado

1. Necesidad de integrar diferentes aplicaciones o servicios.
2. Entorno heterogéneo y trabajo con diferentes tecnologías y estándares.
3. Necesidad de reducir el coste de desarrollo.

Podemos ver reflejadas las situaciones 1 y 2 en nuestro caso, ya que vamos a necesitar integrar diferentes aplicaciones como *Snort* y vamos a trabajar con tecnologías distintas como puede ser CEP o Java.

En el mercado actual existen muchos ESB que pueden cumplir los requisitos necesarios para trabajar con ellos. Algunas de las características que debe tener el ESB que elijamos pueden ser, que sea independiente de la plataforma en la que trabajemos, que posea interoperabilidad entre diferentes tecnologías, sea capaz de soportar normas para servicios web, que posea adaptadores para integrar distintas tecnologías, posibilidad de enriquecer mensajes, que sea capaz de encaminar los mensajes aplicando una política y que tenga la posibilidad de retener los mensajes cuando las aplicaciones no estén disponibles.

Tenemos que tener en cuenta que en el mercado existen tanto ESB comerciales, que no permiten su uso libre, y otros de código abierto. Vamos a centrarnos en estos últimos porque ofrecen todas las características que necesitamos y además tenemos la posibilidad de probarlos y poder elegir el que mejor se adapte a nuestras necesidades. En la tabla 2.4 podemos ver todos los ESB de código abierto que podemos encontrar en el mercado y posteriormente analizaremos los más importantes.

Una vez conocidas todas las alternativas que nos ofrece el mercado, vamos a analizar las más importantes.

Mule ESB [49] desarrollado por *Codehaus [45, 2]* es uno de los más importantes en todo el mercado, es capaz de trabajar con objetos escalables y distribuidos y ofrece una buena interacción entre servicios y aplicaciones que utilicen distintas tecnologías y estándares. Mule tiene como una de sus características más importantes que es bastante ligero y se integra muy bien con aplicaciones Java. Tiene muchas posibilidades de encaminamiento y de administración de mensajes además de un contenedor JBI conocido como *MuleJBI*. Mule se distribuye como un *plugin* para *Eclipse* conocido como *MuleIDE* desde el que se pueden gestionar los proyectos. Además posee un IDE propio llamado *MuleStudio* que aporta una interfaz gráfica más amigable para el desarrollo de proyectos bajo este ESB. Actualmente hay muchas empresas importantes que lo utilizan como por ejemplo *Sony*, *Hewlett-Packard* o *Deutsche Bank* entre otras.

Otra alternativa es *Apache ServiceMix*, que también es uno de los ESB libres más utilizados en el mercado actual. Este ESB implementa el estándar JBI (JSR-208) para lograr un acoplamiento muy bajo entre los diferentes componentes permitiendo así que se pueda adecuar a otro ESB que cumpla con este estándar. En la última versión de este ESB se basa en el estándar OSGi 4.2.0 como modelo de componentización y despliegue de soluciones. El motor de servicios de este ESB permite trabajar con proyectos de Apache como *Camel* que es un encaminador. También permite trabajar con un motor de reglas *Drools [57]* y con *Apache CXF*. Actualmente este ESB es usado en empresas como *British Telecom* o *Cisco Systems*.

Open ESB es otra alternativa a las dos vistas anteriormente, está desarrollado por *Sun Microsystems* y se ofrece como un proyecto de Java.net. Igual que *Apache ServiceMix , Open ESB* está basada en la implementación de la especificación JBI. Tiene funciones similares a *Apache ServiceMi*x aunque en este caso, *Open ESB* se encuentra orientada a aplicaciones basadas en el servidor *Glassfish*.

Una alternativa a los ESB vistos anteriormente es *Apache Synapse*, que realmente es un conjunto de servicios web que se encuentran contenidos en *Apache Axis2* y su objetivo es ofrecer diferentes funcionalidades como pueden ser transformación, encaminadores, validación de mensajes o registros basados en servicios web y en el estándar XML. *Synapse* provee de la abstracción necesaria para trabajar con estándares de servicios web complejos.

JBoss ESB es conocido por su popular herramienta de mapeo objeto-relacional *Hibernate* y por *Seam* que es un framework para construir aplicaciones web 2.0. *JBoss* adquirió en 2006 un ESB conocido como *Rosetta* y ha desarrollado un ESB llamado *JBoss ESB* basado en él. Implementa una completa integración en la que se pueden implementar productos *JBoss* y posee una funcionalidad de encaminamiento basada en el motor de reglas *Jboss Rules*. Este ESB permite trabajar con componentes EJB y POJO.

Otro ESB que debemos nombrar es *OW2 PEtALS*, que posee una arquitectura orientada a trabajar en entornos distribuidos donde se pueden integrar diferentes instancias de *PEtALS* en un sólo entorno del ESB. Así podemos acceder desde diferentes lugares a un servicio que se encuentre almacenado en otra máquina.

Apache Tuscany es una aplicación que no puede ser definida como un ESB al uso, pero que ofrece una serie de funcionalidades que nos permite trabajar con él como tal. *Apache Tuscany* es una implementación de una arquitectura de servicios orientados a componentes. Esta especificación define un lenguaje (SCDL) con el cual se pueden definir servicios que se pueden comunicar con otros componentes y servicios.

Criterio	Mule	ServiceMix	Open ESB
Funcionalidades más importantes	4	4	3
Documentación	4	3	4
Visibilidad en el mercado	5	4	3
Comunidad y soporte	5	4	3
Flexibilidad y personalización	5	4	3
Protocolos de transporte y conectividad	4	4	3
Capacidad de integración	5	5	3
Productividad con un IDE	4	4	5

Criterio	Synapse	PEtALS
Funcionalidades más importantes	4	4
Documentación	4	3
Visibilidad en el mercado	3	3
Comunidad y soporte	4	4
Flexibilidad y personalización	5	4
Protocolos de transporte y conectividad	3	4
Capacidad de integración	4	4
Productividad con un IDE	3	4

Cuadro 2.4: Comparativa ESB

El último ESB del que debemos hablar es *Spring Integration*, desarrollado por *SpringSource* que es la compañía que el conocido framework *Spring*. Este ESB ofrece apoyo para la integración de proyectos con plantillas JMS y servicios web. *Spring* ofrece funcionalidades de integración basadas en patrones que se pueden encontrar en el libro *Enterprise Integration Patterns*.

Para facilitar la elección de un ESB vamos a observar la tabla 2.4 donde se observa una comparativa entre los mejores ESB libres. El criterio de valoración en cada caso ha sido de 5 la máxima puntuación y 1 la mínima. Si observamos los resultados vemos que tanto *Mule* como *ServiceMix* son las mejores soluciones que ofrece el mercado en la actualidad.

Una vez que hemos visto todos los ESB que ofrece el mercado y una comparativa de ellos, tenemos que decantarnos por la opción que mejor se ajuste a nuestras necesidades. En este caso vamos a utilizar *Mule ESB*. Ya que nos

va a aportar todos los componentes que necesitamos para poder integrar las diferentes tecnologías, además de que posee una interfaz gráfica de usuario o *Graphical User Interface* GUI como es *Mule Studio* que nos va a facilitar el desarrollo de nuestro proyecto. En la figura 2.5, extraída de [49], podemos ver algunas de las funcionalidades que nos ofrece Mule.

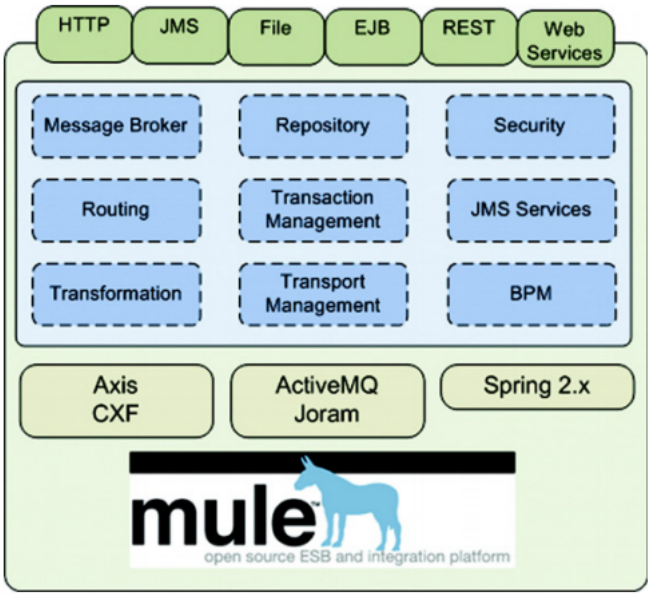


Figura 2.5: Resumen de funcionalidades de Mule ESB

Podemos ver como Mule permite trabajar con servicios web, con estándares como HTTP o REST e incluso con ficheros, algo que nos será muy útil en nuestro proyecto para poder hacer uso de los ficheros de alerta que nos genera *Snort*. También podemos observar algunas funcionalidades importantes de Mule como el gestor de mensajes, la seguridad, el gestor de transacciones, sus encaminadores o la capacidad para realizar transformaciones.

En el anexo correspondiente 11.2 al manual de usuario se pueden ver más a fondo como se puede configurar este ESB y cuáles son los componentes disponibles para desarrollar un proyecto.

2.5. CEP

El procesamiento de eventos complejos [41, 51, 61, 63] es una tecnología emergente orientada a las aplicaciones dirigidas por eventos y que actualmente poseen una gran importancia. Esta tecnología tiende a utilizarse cada día más, ya que provee de las características necesarias a los sistemas para tomar decisiones en tiempo real. En nuestro proyecto vamos a hacer uso de ella para detectar las amenazas a partir de los eventos generados por *Snort*. Para comprender bien cuál es el funcionamiento de esta tecnología y como la vamos a aplicar tenemos que conocer una serie de conceptos previos.

Un evento es algo que ocurre, ha ocurrido o se espera que ocurra ya sea dentro del sistema o fuera de él. Estos eventos pueden tener tres tipos de relaciones. Una basada en la temporalidad del evento, es decir, si el evento incorpora información respecto a la fecha o a la hora en la que se ha producido. Otra relación es la de causalidad que se da en el momento en el que un evento sucede debido a que cierto evento concreto ha ocurrido con anterioridad. Por último tenemos una relación de agregación, esta sucede cuando una actividad consiste en la unión de varios eventos. De esto podemos concluir que los eventos tienen asociados una serie de datos, de operaciones y de relaciones entre ellos. En el caso particular de la seguridad, cada alerta que se produzca en el sistema será un evento que tendrá asociado un componente temporal que indicará el que se ha producido, una serie de datos relativos al ataque además de una relación entre ellos, ya que en muchas ocasiones un evento vendrá determinado por otro que se ha producido anteriormente. A partir de ahí nuestro trabajo consistirá en detectar ataques en los eventos complejos a partir de estos eventos simples. Podemos ver de manera más clara en la figura 2.6 un posible ataque distribuido de denegación de servicio en la que dos máquinas distintas atacan a la nuestra y gracias a CEP detectamos el evento complejo referente a este posible ataque.

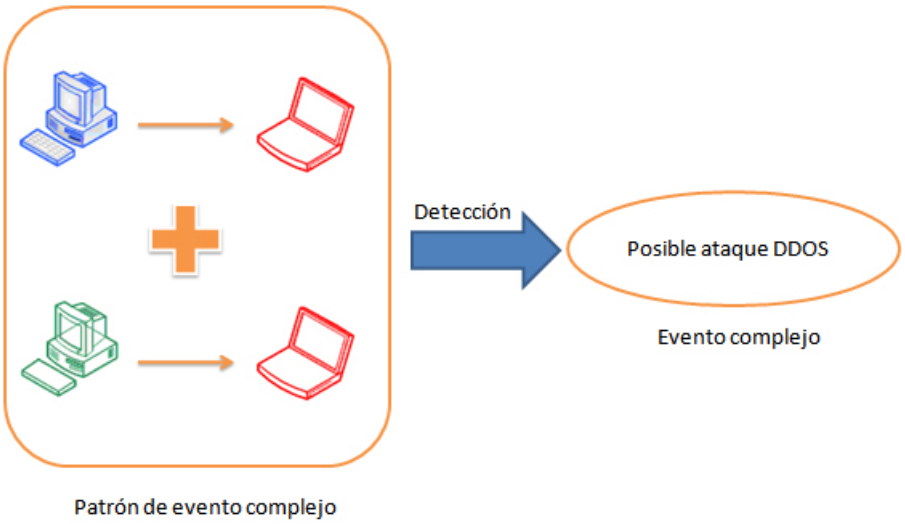


Figura 2.6: Patrón de evento complejo

Otro concepto que debemos conocer es el de flujo de evento que es una secuencia de eventos ordenados en el tiempo. Existen dos tipos de flujos de eventos según su tipo, si los eventos son del mismo tipo estamos ante un flujo de eventos homogéneo, en cambio si los eventos del flujo son de distinto tipo el flujo es heterogéneo. En este punto la función de CEP es filtrar los eventos que no son importantes y reconocer los patrones que sí lo son. Estos patrones los habremos diseñado nosotros previamente, gracias a lo cual podremos detectar eventos complejos, también llamados situaciones. Por lo tanto el objetivo final va a ser detectar eventos para observar las relaciones que existen entre ellos y obtener así situaciones complejas, que en nuestro caso serían ataques detectados a nuestro sistema.

Gracias a esto CEP nos permite trabajar en tiempo real, ya que se los eventos se detectan instantáneamente y el tiempo de reacción es el mínimo posible. En el caso de nuestro sistema esto es un punto de vital importancia, ya que cuando hablamos de sistemas de seguridad y de ataques, el tiempo es un factor crítico.

Una vez que hemos visto las características que nos ofrece CEP, podemos plantearnos en qué sistemas es adecuado su uso. CEP puede aportarnos las siguientes ventajas [51]:

- Nos aporta escalabilidad. Podemos extender aplicaciones existentes de manera flexible.
- Permite la separación de la lógica del procesamiento de eventos respecto del resto de la aplicación.
- Permite a la aplicación recibir grandes cantidades de datos y ofrece una salida al usuario.
- Ofrece la capacidad de trabajar en tiempo real.
- Permite reducir el coste de una aplicación ya que se ofrece una abstracción para manejar los eventos, disminuyendo así los costes de desarrollo y mantenimiento.

Una parte importante de CEP que hay que tratar es la referente a los patrones. Éstos son los que esta tecnología utiliza para detectar los eventos complejos dentro de un flujo de eventos. En el momento en el que se detecte un patrón, ésta información se enviará al motor de CEP. Es importante tener en cuenta que estos patrones tienen asociada información relativa al tiempo en el que se ha producido, permitiendo trabajar así conociendo que eventos son posteriores a otros.

A la hora de realizar la clasificación de los patrones hacerlo de diferentes maneras. Paschke [61] propone una clasificación en base a cuatro puntos que vemos a continuación.

- **Soluciones buenas y malas**. En este punto distingue cuáles son los patrones que ofrecen una solución a problemas frecuentes y cuáles producen consecuencias negativas. Estos últimos son conocidos como antipatrones.
- **Niveles de abstracción**. Se distinguen los siguientes niveles de abstracción:

  - Instrucciones y buenas prácticas.
  - Patrones de gestión.
  - Patrones de arquitectura.
  - Patrones de diseño.
  - Patrones de mapeo.
  - Patrones de ejecución.
  - Patrones de refactorización.
  - Refactorización.

- **Objetivos a alcanzar**. Se refiere a que tipo de problema debe proporcionar solución el patrón:

  - Patrones de adopción.
  - Patrones de negocios.
  - Patrones de integración.
  - Patrones compuestos.
  - Patrones de flujos de trabajo.
  - Patrones de coordinación.
  - Patrones personalizados.
  - Patrones de aplicación.

- **Nivel de gestión**. En este punto se describen las decisiones de diseño a nivel operacional, táctico y estratégico de la gestión de servicios CEP. Podemos distinguir tanto patrones estratégicos, como tácticos y operacionales.

En cuanto a las estructuras de los patrones, este mismo autor propone la siguiente:

- Nombre del patrón.
- Descripción del problema.
- Otros nombres del patrón.

- Descripción del escenario del problema y propuesta de solución.
- Situaciones y condiciones en las que el patrón puede aplicarse.
- Descripción del diseño de la estructura.
- Clases y objetos que describen el patrón de diseño.
- Ventajas e inconvenientes del patrón.
- Posibles variaciones del patrón y su descripción.
- Ejemplos de aplicaciones en los que se utiliza el patrón.
- Patrones relacionados que describen tareas parecidas.

Actualmente existen diferentes herramientas CEP. En la tabla 2.5 podemos ver qué herramientas comerciales existen y cuáles son libres. En nuestro caso vamos a utilizar una solución libre, concretamente Esper.

2.5.1. Esper

Una vez analizadas todas las herramientas que hemos visto en la tabla para implementar CEP, nosotros hemos decidido trabajar con Esper que es un motor CEP desarrollado por la compañía *EsperTech Inc.* y es distribuido mediante licencia GPL.

Como podemos ver en la figura 2.7 la arquitectura de Esper está formada por diferentes componentes. Dentro de este contenedor debemos diferenciar las tres partes principales que lo componen, los eventos, los patrones de eventos y por último las interfaces que notifican eventos, conocidos como *listeners*.

El funcionamiento de este motor es el de analizar un flujo entrante de eventos y filtrar únicamente los eventos que tienen algún interés y puedan ser relacionados con los patrones de eventos que se hayan diseñado. La entrada de eventos puede estar implementada en diferentes formatos como pueden ser objetos planos de java o *Plain Old Java Object* (POJO), documentos XML y *map* de java. En nuestro proyecto vamos a trabajar con objetos POJO.

Cuadro 2.5: Herramientas CEP

Tipo	Nombre	Desarrollador	Ref.
Comercial	RulePoint	Agent Logic	[38]
	Event Zero	Event Zero	[52]
	Amit	IBM	[37]
	Active Middleware Technology	IBM	[53]
	InfoSphere Streams	IBM	[54]
	WebSphere Business Events	IBM	[55]
	Microsoft CEP server	Microsoft	[39]
	Autopilot M6 CEP	Nastel Technologies Inc.	[59]
	Oracle CEP 10g	Oracle	[60]
	Progress Apama	Progress Software	[62]
	RealTime Monitoring	Realtime Monitoring GMBH	[64]
	ruleCore CEP Server	Rulecore	[65]
	Starview Event Servers	Starview Technology	[69]
	StreamBase	StreamBase Systems	[70]
	Sybase Aleri Streaming Platform	Sybase Inc.	[71]
	TIBCO BusinessEvents	Tibco	[72]
	Truviso	Truviso Inc.	[73]
	UC4 V8	UC4	[75]
	Vantify	WestGlobal Ltd.	[78]
Libre	Etalis	D. Anicic y P. Allen	[40]
	Aurora	Brandeis University, Brown University y MIT	[42]
	Borealis	Brandeis University, Brown University y MIT	[43]
	Medusa	Brandeis University, Brown University y MIT	[44]
	Intelligent Event Processor	CollabNet Inc.	[46]
	Cayuga	Cornell University	[47]
	Esper/NEsper	EsperTech Inc.	[50]
	Drools Fusion	JBoss Community	[57]
	PADRES	University of Toronto	[58]
	Rapide	Stanford University	[67]
	STREAM	Stanford University	[68]
	Sopera	SOPERA GmbH	[66]
	TelegraphCQ	UC Berkeley	[74]
	PIPES	University of Marburg	[76]
	SASE	UC Berkeley / University of Massachusetts Amherst	[77]

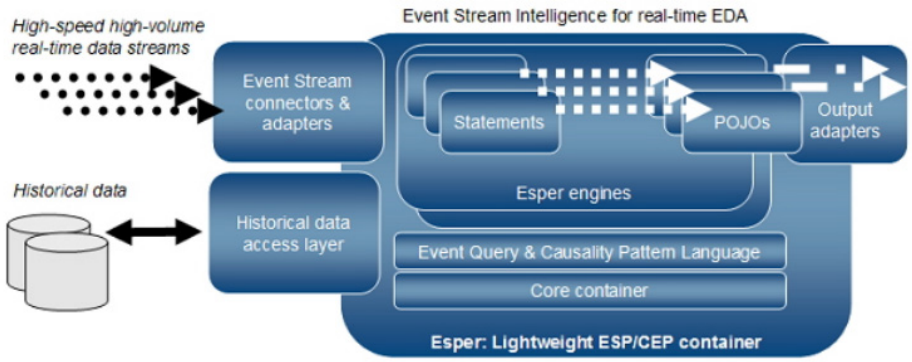


Figura 2.7: Arquitectura de Esper

Uno de los puntos más importantes de los que debemos hablar es de la manera de diseñar los patrones, para ello Esper posee un lenguaje parecido a SQL llamado lenguaje de procesamiento de eventos o *Event Processing Lenguaje* (EPL) que nos va a permitir indicar ventanas temporales en las que poder filtrar eventos, trabajar con los flujos creando algunos nuevos a partir de los ya existentes y diseñar patrones que se ajusten a nuestra lógica de negocio.

La manera en la que trabaja Esper con estos patrones es la de un funcionamiento continuo, es decir, si tenemos un flujo de eventos, las sentencias EPL están continuamente siendo evaluadas a la espera de poder identificar un patrón. Cada vez que llega un evento a través del flujo, se compara con todos los patrones activos y se notifica a los *listener* cuando se obtiene una relación entre un evento y un patrón. Esto permite que entre los eventos y sus *listener* no haya acoplamiento, por lo tanto un patrón puede tener uno o varios *listeners* que realicen ciertas acciones concretas cuando reciban una notificación.

A la hora de trabajar con patrones y con expresiones EPL Esper permite hacerlo en dos modos distintos. Uno de ellos es el modo *push*, en este caso cada vez que se identifica un evento con un patrón Esper lo notifica al *listener* correspondiente. Es decir, el orden sería el siguiente, primero se recibe un evento en el motor de CEP al que posteriormente se le van a aplicar todos los patrones y en los casos necesarios se le notificará al *listener* correspondiente. El otro modo de trabajar es el conocido como modo *pull*, en este caso se

realizan consultas constantes al motor para conocer los eventos que se relacionan con los patrones que hemos diseñado. Es decir, a diferencia del modo *push*, en este caso también se enviarían primero los eventos al motor pero posteriormente lo que se hace es calcular las consultas EPL y por último se obtiene el resultado de aplicar dicha consulta.

Otro concepto importante que tenemos que conocer de Esper son los tipos de ventana que existen. Las ventanas temporales son la representación de un conjunto de eventos que van variando a lo largo del tiempo y con la aparición de nuevos eventos. Podemos distinguir dos tipos dentro de ellas.

- **Ventanas deslizantes**: En este caso se van moviendo paso a paso hacia delante cada vez que un nuevo evento entra a formar parte de la ventana. En este caso un evento puede formar parte de la misma ventana a lo largo de varias iteraciones. Puede verse en la figura 2.8 basada en [56]
- **Ventanas abatibles**: En este caso la ventana no avanza hasta que no se complete, es decir, se mueve también hacia adelante en incrementos que corresponden a su longitud. A diferencia de las ventanas deslizables, un evento sólo forma parte de una ventana en una iteración. Para verlo mejor ver la figura 2.9 basada en [56]

También podemos realizar una clasificación en las ventanas según si son por lotes o no. En el caso de que sí lo sean esperan a que la ventana se llene para poder lanzar un evento, mientras que en las ventanas que no son por lotes cada vez que se produce un cambio se lanza un evento.

Es muy importante conocer como trabaja Esper a la hora de detectar eventos. Cuando se produce un evento o varios de ellos y se identifican con un patrón que hemos diseñado Esper detecta un en evento. Es importante conocer que el factor del tiempo está muy presente aquí, ya que los patrones pueden estar basados en él. Una expresión en EPL de un patrón está formada por átomos de patrones y sus operadores. Vamos a hablar sobre estos átomos que se pueden clasificar en tres categorías.

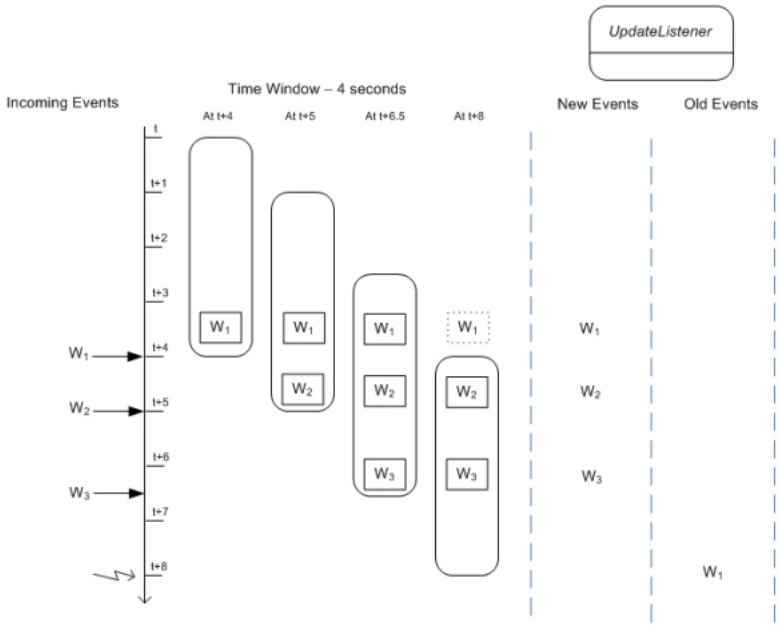


Figura 2.8: Ventanas deslizantes de Esper

- **Expresiones de filtro**. Sirven para especificar un evento en especial que tenemos que buscar. La estructura es la siguiente, se escribe el nombre del tipo de evento y posteriormente entre paréntesis los criterios por los que se va a realizar el filtrado. Se pueden utilizar operadores lógicos como *and* que puede ser sustituido por una coma.

- **Observadores de evento**s. Sirven para especificar intervalos de tiempo o estrictamente un tiempo determinado. En el caso de que queramos especificar un intervalo se va a esperar ese tiempo antes de que el valor del observador sea cierto. Se puede especificar un periodo de tiempo en días, horas, minutos, segundos y milisegundos. En el caso de un tiempo específico utilizamos la expresión *timer:at* seguida por los minutos, horas, días del mes, meses y días de la semana, y opcionalmente segundos. La expresión se evaluará verdadera cuando se alcance ese tiempo determinado.

- **Observadores personalizados**. Son los que observan los eventos externos que no estén controlados por el motor.

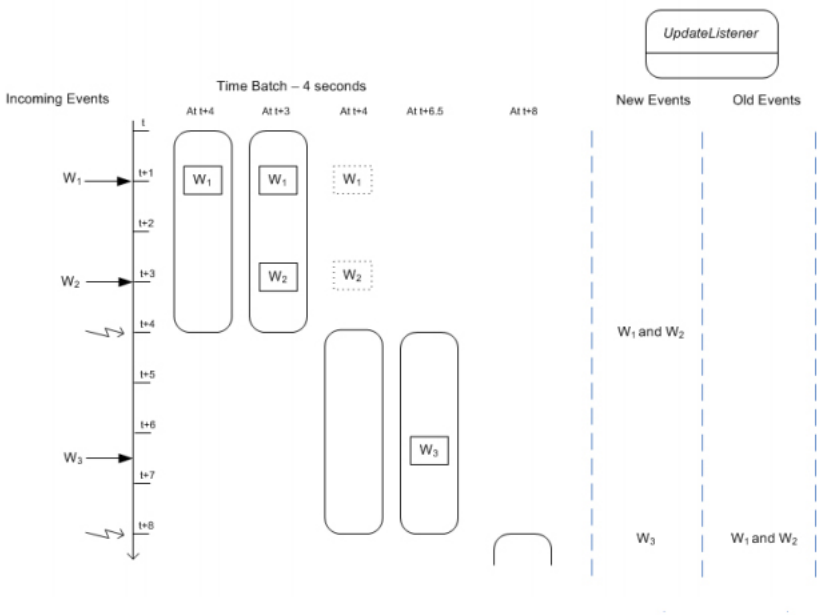


Figura 2.9: Ventanas abatibles de Esper

Una vez vistos estos átomos de patrones, tenemos que ver cuales son los operadores que le podemos aplicar. Tenemos cuatro tipos de operadores y se utilizan para controlar el ciclo de vida de las expresiones y para combinar estos átomos de manera lógica o temporal. Debemos indicar que podemos utilizar varios de estos operadores en un mismo patrón además de poder anidarlos mediante paréntesis.

- **Operadores temporales**. Estos operadores actúan sobre el orden de los eventos e indican que es estrictamente necesario que la expresión de la izquierda sea verdadera para que se pueda evaluar la expresión de la derecha. El operador se indica mediante una flecha $\rightarrow$.
- **Operadores de repeticiones**. Van a controlar las repeticiones de las subexpresiones de los patrones. Vamos a utilizar la palabra reservada *every*, y cuando la expresión que sigue a esta palabra sea evaluada se debe reiniciar la subexpresión del patrón. La diferencia de utilizar *every* o no hacerlo radica en que si especificamos un evento A, cuando se encuentre la primera relación se finaliza la búsqueda, en cambio si

indicamos *every A* esta búsqueda no va a finalizar. Podemos construir operaciones compuestas junto al operador temporal. Dentro de estos operadores de repetición tenemos otro que utiliza la expresión *every-distinct*. La diferencia respecto a *every* es que se van a eliminar los resultados duplicados. Otros operadores de repetición son *[num]*, en este caso de debe evaluar como verdadera la expresión ese número de veces determinado. También existe la expresión *until* que va a buscar eventos de un tipo hasta que lleguen eventos del tipo que indiquemos a la derecha del operador.

- **Operadores lógicos**. En este caso tenemos los mismos operadores lógicos que ofrecen la mayoría de lenguajes de programación. Podemos hacer uso del operador *and* que necesita que ambas expresiones sean evaluadas como verdaderas para que se evalue como verdadera la expresión completa. El operador *or* que sólo necesita que se evalúe como verdadera una de las dos expresiones que componen la expresión completa y el operador *not* que niega el valor de la expresión a la que se le aplique dicho operador.

- **Guardas**. Son operadores destinados a controlar el ciclo de vida de las subexpresiones mediante condiciones *where* de la misma manera que se hace en SQL. Tenemos el operador *timer:within* que especifica un intervalo de tiempo en el cual debe evaluarse como verdadera la expresión, de no ser así se le asignará un resultado de expresión falsa. El operador *timer:withinmax* se diferencia del anterior en que también se le aplica un contador que va a contar el número de relaciones. La expresión va a finalizar cuando o bien se alcance el tiempo determinado o llegue al número de repeticiones indicadas. Por último tenemos el operador *while* que va ligada a una expresión que debe ser evaluada para cada relación. Esta subexpresión finalizará cuando la expresión del *while* sea evaluada como falsa. Por supuesto podemos crear guardas personalizadas y combinar varias de ellas.

Se pueden relacionar los patrones cuando se detecta una secuencia de eventos. Estos resultados se podrán procesar posteriormente.

2.6. SOA 2.0 + CEP

Ya hemos visto las tecnologías y arquitecturas que vamos a utilizar a lo largo del desarrollo de este proyecto y cual es su situación actual en el mercado. Por último nos falta ver la unión de las arquitecturas SOA con el enfoque de las tecnologías CEP.

En el mercado actual existen bastantes más aplicaciones basadas en arquitecturas SOA que en CEP, esto es debido principalmente a que CEP es una tecnologías bastante novedosa mientras que SOA se lleva utilizando más tiempo. También hay que indicar que en las arquitecturas SOA los usuarios deben conocer bien los servicios con los que van a interaccionar, como operan y cual es su interfaz para poder realizar la comunicación, en CEP esto es distinto ya que la orientación a eventos crea sistemas con menos acoplamiento dónde hay unos sensores o fuentes que generan la información y van a ser los consumidores los que se encarguen de obtener y tratar estos datos.

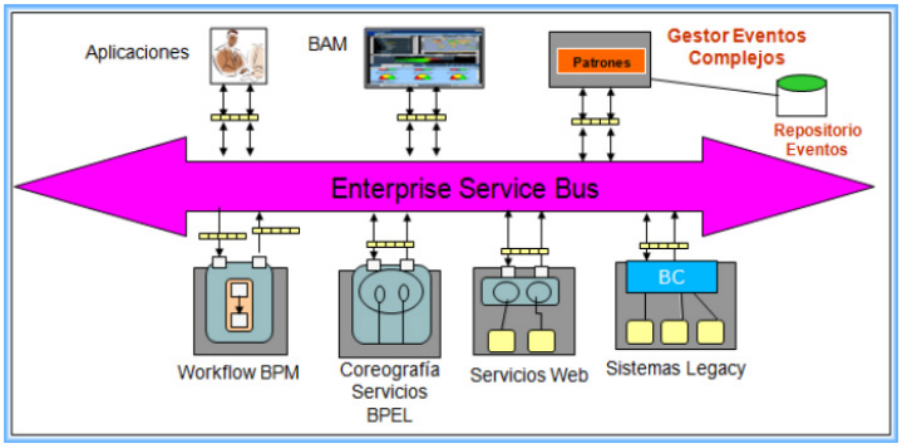


Figura 2.10: Arquitectura ESB y CEP

También es cierto que presentan varias similitudes como pueden ser la generación de sistemas flexibles, que presentan bajo acoplamiento y que son modulares. Hemos visto que la manera de unir estas tecnologías vamos a realizarlo a través de un ESB. Autores como Ayllón y Reina [41] han definido

los elementos que aparecen en este tipo de arquitecturas. En la figura 2.10, propuesta en [41], podemos ver los diferentes sistemas de comunicación que se establecen.

Entre todas estas aplicaciones podemos destacar los servicios web que proporcionarán funcionalidades al ESB, las aplicaciones para que el usuario pueda realizar los diferentes trabajos o el gestor de eventos complejos donde se van a procesar en tiempo real los eventos que se produzcan. En este punto se realiza la integración de CEP y SOA.

Gracias a estas estructuras se van a reducir el número de conexiones punto a punto y se realizará un enrutamiento de mensajes a aplicaciones de manera confiable. Además existen adaptadores para trabajar con diferentes sistemas de mensajes como JMS, HTTP/SOAP, FTP y SQL. A la hora de realizar la comunicación, ésta se podrá realizar punto a punto mediante colas de mensajes o en formato publicador/subscriptor.

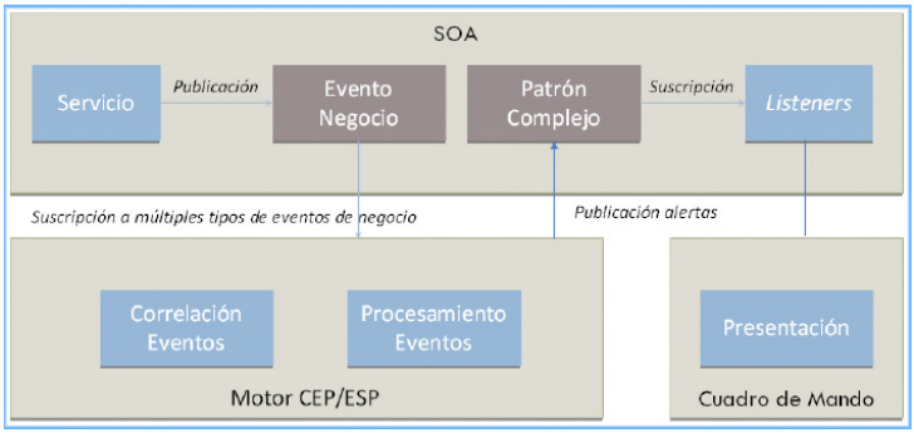


Figura 2.11: Integración del motor CEP en SOA

A la hora de definir un patrón, estos mismos autores definen la manera en la que integrar el motor de CEP a SOA. En la figura 2.11 podemos ver como el motor va a recibir los diferentes tipos de eventos que se generen y publique el servicio correspondiente. Posteriormente estos eventos se deben procesar

y encontrar las relaciones que existan entre ellos, detectando así los patrones complejos. Una vez encontrados, se enviará la información a los *listeners*, que procesarán dicha información y presentarán los resultados al usuario final.

*

Capítulo 3

Desarrollo del calendario

A la hora de desarrollar este sistema hemos seguido un modelo de ciclo de vida incremental que es el que más se ajusta a nuestro proyecto y el más adecuado debido a las características que presentaba. Este modelo de ciclo de vida es el correcto ya que al tener un fuerte componente de investigación, todos los requisitos no estaban claros desde un principio ya que podíamos toparnos con ciertas dificultades a la hora de realizar alguna de las integraciones de las diferentes tecnologías.

Este modelo es iterativo y se realizan diferentes ciclos en los que se van realizando un incremento que nos va acercando cada vez más al producto final.

Vamos a exponer las diferentes etapas que se han seguido para construir este proyecto, teniendo en cuenta que no es necesario finalizar una para poder comenzar con la siguiente.

3.1. Etapas

Vamos a especificar las tareas que se han realizado en las diferentes etapas expuestas en orden cronológico. Posteriormente mediante un diagrama de Gantt se verá de manera más clara la duración de cada una de estas etapas.

3.1.1. Primera iteración: conocimientos preliminares

En esta primera fase se incluyen las reuniones con el tutor del proyecto donde se comenzó a ver cuáles serían los objetivos del sistema a construir, qué tecnologías y herramientas se podrían utilizar y una primera visión sobre la utilidad de estas nuevas tecnologías. En esta fase se incluyen las primeras investigaciones bibliográficas sobre los conceptos como las arquitecturas que se planeaban seguir, los lenguajes de programación o los sistemas de seguridad.

También se vio algún sistema real que se ha desarrollado utilizando el procesamiento complejo de eventos y se ha realizado la elicitación de requisitos.

3.1.2. Segunda iteración: investigación

Dado la cantidad de conceptos y enfoques novedosos en el proyecto, el periodo de investigación comenzó con la búsqueda de recursos bibliográficos sobre las arquitecturas SOA y EDA y su integración en arquitecturas SOA 2.0. Este periodo de investigación se alargó durante gran parte del desarrollo del proyecto solapando en el tiempo a otras fases, ya que a medida que se avanzaba en el desarrollo era necesaria la investigación sobre ciertas áreas del proyecto.

En este punto se investigó sobre los diferentes ESB del mercado, probando varios de ellos hasta elegir Mule ESB, también se investigó sobre diferentes opciones para la obtención de información de los ataques, sobre el funcionamiento y configuración de *Snort* y posteriormente sobre CEP y los diferentes motores que podíamos utilizar hasta escoger Esper. También se llevó a cabo el estudio sobre el lenguaje EPL para la construcción de los patrones de eventos.

Otro de los puntos sobre los que se ha investigado han sido los sistemas de seguridad, lo que había en el momento del comienzo del proyecto, las alternativas y la mejor opción para el desarrollo. En este campo también se han investigado cuáles son los ataques más usuales, como se realizan y como pueden ser detectados.

3.1.3. Tercera iteración: Snort

En esta etapa y a medida que se producían los resultados de la investigación en este concepto, se realizó la instalación de *Snort* y de un paquete de reglas adecuado para el uso que le vamos a dar. También se realizó la configuración de este software y se llevaron a cabo varias pruebas de ataques sobre el sistema para comprobar el funcionamiento de *Snort* y las diferentes alertas que podía generar.

3.1.4. Cuarta iteración: ESB

En este punto y una vez que en el periodo de investigación se eligió Mule como ESB se realizó la instalación de Mule ESB, su integración con Eclipse y el desarrollo de algunas pruebas para comprobar su potencia. Posteriormente se instaló y configuró MuleStudio que ha sido la herramienta con la que finalmente hemos trabajado.

Hasta este punto no se había realizado ninguna integración entre tecnologías, simplemente se habían analizado y probado por separado para posteriormente realizar dicha integración.

3.1.5. Quinta iteración: integración de Snort con Mule ESB

Esta es una de las iteraciones críticas ya que una vez elegidas y probadas estas dos tecnologías por separado llegaba el momento de unirlas para lograr que trabajasen juntas. En este punto se realizó la integración de manera que el ESB estuviese continuamente pendiente de las alertas que generaba *Snort* para tratarlas inmediatamente, realizando así una correcta integración y consiguiendo que la primera parte del proyecto trabajase en tiempo real dado que en el momento en el que se generaba una alerta, ésta pasaba directamente al ESB para que fuese procesada.

3.1.6. Sexta iteración: procesamiento de las alertas

Una vez que las alertas generadas por *Snort* llegaban al ESB debíamos tratarlas, obtener sólo la información relevante y transformarlas a eventos para la posterior fase del procesamiento de eventos complejos. En este punto se utilizaron algunas herramientas del ESB como los transformadores para manejar los diferentes tipos de mensajes y se desarrollaron las clases necesarias en Java para discriminar la información y crear eventos sólo con los datos importantes para tratar las amenazas.

3.1.7. Séptima iteración: integración de CEP y Mule ESB

En esta fase se llevó a cabo la integración del motor de CEP, Esper, con Mule ESB, logrando así que la información fluyese en tiempo real por el ESB y se enviara a Esper para la detección de los diferentes ataques.

En este punto teníamos construido un sistema que conseguía en tiempo real recibir alertas producidas por *Snort* sobre posibles ataques, su envío al ESB donde se trataban y generaban los eventos y el envío de estos al motor de CEP.

3.1.8. Octava iteración: diseño de patrones de eventos en EPL

Esta etapa iba bastante ligada al periodo de investigación, ya que lo primero fue la elección de los ataques que se iban a detectar, utilizando criterios para su elección como la importancia de los ataques, la frecuencia con la que se realizaban y la información de la que disponíamos en el momento de recibir una alerta.

Se decidió que el sistema detectaría ocho posibles ataques y se implementaron sus patrones mediante el lenguaje de programación EPL. Estos patrones van muy relacionados con el motor de CEP ya que ahí es donde se van a comparar las alertas que llegarán en el flujo de eventos con los patrones de eventos que hemos diseñado.

3.1.9. Novena iteración: publicación de los ataques

En este punto, ya habíamos conseguido identificar cuándo era atacado nuestro sistema y debíamos elegir la forma de la que nuestra aplicación iba a reaccionar. Aquí se implementaron los *listeners* correspondientes a cada uno de los patrones de eventos y la publicación de la información relativa a estos ataques que se detectaban.

También se desarrolló en Java las clases necesarias para el envío de esta información por correo electrónico al responsable de la seguridad del sistema y la integración de estas clases con los *listeners* de los patrones.

3.1.10. Décima iteración: pruebas

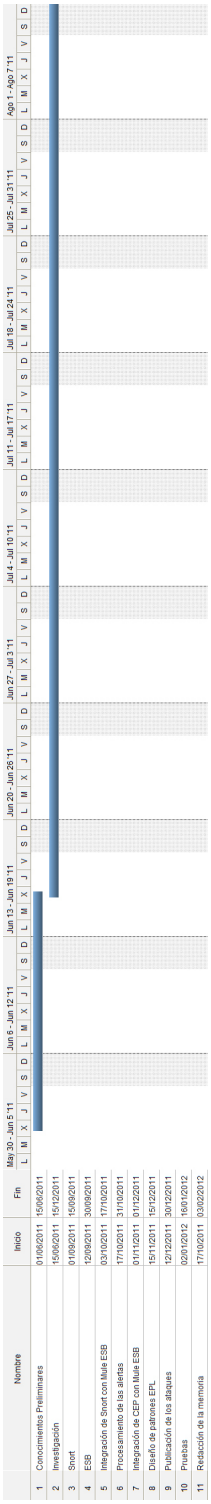
A lo largo de esta iteración se realizaron las diferentes pruebas para comprobar la integridad del sistema y el correcto funcionamiento de todos los módulos que lo componen. Se realizaron pruebas sobre todos los tipos de ataques y la respuesta que ofrecía el sistema, obteniendo resultados positivos en todas ellas.

3.1.11. Undécima iteración: redacción de la memoria.

Si bien hemos colocado esta etapa en último lugar, al igual que la iteración relativa al periodo de investigación, ésta se ha llevado a cabo a lo largo de gran parte del desarrollo del proyecto. La redacción de la memoria comenzaba en el momento en el que tanto las tecnologías como la arquitectura definitiva a utilizar estaban claras, para continuar con esta etapa hasta que concluía en último lugar.

3.2. Diagrama de Gantt

Para reflejar la distribución de las diferentes etapas a lo largo del tiempo hemos diseñado el diagrama de Gantt que se puede ver en las figuras 3.1, 3.2 y 3.3. Este diagrama se ha realizado con la herramienta libre Gantter que se puede encontrar en [7].



	Nombre	Inicio	Fin
1	Conocimientos Preliminares	01/06/2011	15/06/2011
2	Investigación	15/06/2011	15/12/2011
3	Snort	01/09/2011	15/09/2011
4	ESB	12/09/2011	30/09/2011
5	Integración de Snort con Mule ESB	03/10/2011	17/10/2011
6	Procesamiento de las alertas	17/10/2011	31/10/2011
7	Integración de CEP con Mule ESB	01/11/2011	01/12/2011
8	Diseño de patrones EPL	15/11/2011	15/12/2011
9	Publicación de los ataques	12/12/2011	30/12/2011
10	Pruebas	02/01/2012	16/01/2012
11	Redacción de la memoria	17/10/2011	03/02/2012

Figura 3.1: Diagrama de Gantt 1/3

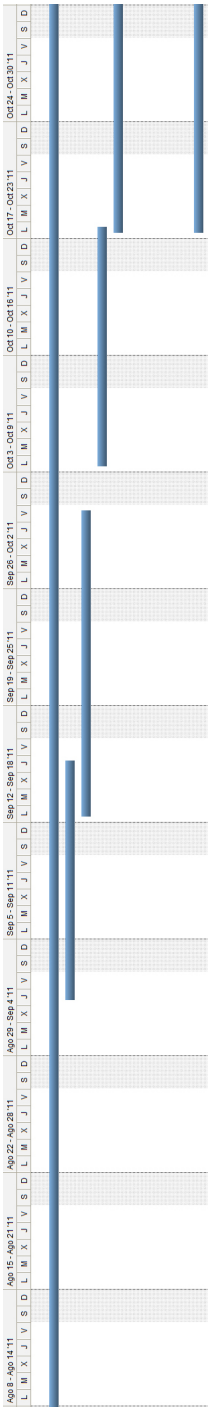


Figura 3.2: Diagrama de Gantt 2/3

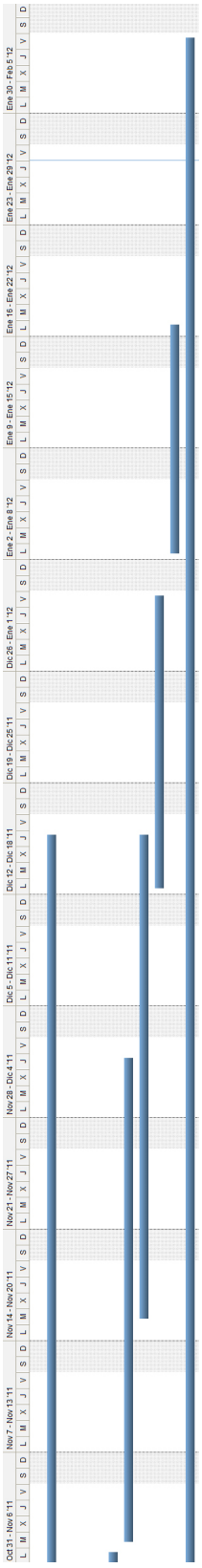


Figura 3.3: Diagrama de Gantt 3/3

Capítulo 4

Descripción general del proyecto

4.1. Perspectiva del producto

4.1.1. Entorno del producto

Este proyecto no es la continuación de ningún otro, sino que su desarrollo a comenzado desde la base. Podemos enmarcar dicho proyecto en el área de los sistemas de seguridad informática, más concretamente en los detectores de intrusos como hemos visto en la sección 2.1.

Para el desarrollo del proyecto sólo partimos del uso de un *sniffer* como es *Snort* que nos servirá como sensor de alertas y a partir de las cuáles trabajaremos.

Como hemos indicado anteriormente el uso de la aplicación creada en este proyecto no excluye del uso de otras herramientas para aumentar la seguridad de los equipos informáticos, sino que es altamente recomendable el uso de otras aplicaciones para poder incrementar dicha seguridad.

4.1.2. Interfaz de usuario

El software que hemos desarrollado no presenta interfaz gráfica. Para trabajar con esta herramienta se puede hacer desde la línea de comandos o bien desde

la propia interfaz de MuleStudio.

Para trabajar con *Snort* también debemos usar la consola de comandos, aunque existen algunos paquetes que permiten trabajar con una interfaz gráfica. En nuestro caso esto no es necesario ya que la única función de *Snort* que vamos a utilizar es su trabajo como *sniffer*, que se dedicará a monitorizar el tráfico de red de manera permanente y a generar las alertas correspondientes.

4.2. Funciones

Las funciones que presenta nuestra aplicación son las siguientes:

- Monitorización del tráfico de red.
- Generación de alertas relacionadas con posibles intentos de intrusión en el sistema.
- Procesamiento y transformación de alertas en eventos y filtrado de la información.
- Capacidad de filtrado de los eventos no relacionados con los ataques incluidos en la aplicación.
- Detección de ataques al sistema mediante procesamiento de eventos complejos.
- Envío de la información relativa a los ataques por email en tiempo real.

4.3. Características de los usuarios

Ya hemos visto que esta aplicación está destinada a ser usada para aumentar la seguridad de los equipos informáticos que se encuentran trabajando en una red y su objetivo es la detección de amenazas que pongan en riesgo la integridad de la información almacenada en el equipo evitando posibles intrusiones.

Este software puede ser utilizado tanto en empresas como por usuarios particulares y va destinado principalmente a dos tipos de usuarios bien diferenciados.

En primer lugar, la persona que va a usar este software no necesita tener conocimientos de seguridad ni ningún conocimiento avanzado de informática, ya que una vez que se lanza el software no hay que hacer nada, debido a que automáticamente se dedica a escanear la red y realiza todo el proceso de detección de ataques de manera automática sin la necesidad de intervención del usuario.

El otro usuario al que va dirigida la aplicación es el destinatario de la información referente a los ataques, es decir, si el software se encuentra en una empresa esta información iría dirigida al responsable de la seguridad informática. Como es lógico este otro usuario si debe tener conocimientos sobre seguridad, ya que será él el encargado de actuar y utilizar las diferentes herramientas a su alcance para proteger el sistema una vez que nuestra aplicación lo ha alertado indicándole en tiempo real el ataque que se está produciendo y todos los datos referente a él.

4.4. Restricciones generales

4.4.1. Control de versiones

Como hemos indicado en el capítulo 3 vamos a seguir el modelo de ciclo de vida incremental, por lo tanto vamos a necesitar un sistema para poder controlar las versiones de nuestro desarrollo.

Gracias a estos sistemas vamos a poder almacenar de manera segura todas las versiones que vayamos desarrollando del proyecto, pudiendo volver a una anterior en cualquier momento y asegurándonos de que no vamos a perder la información en caso de que ocurra cualquier problema en nuestro equipo.

En nuestro caso vamos a utilizar *Subversion* [25], que se distribuye de manera libre gracias a su licencia Apache/BSD. Esta herramienta tiene como diferencia frente a otras es que las diferentes versiones que se van subiendo no tienen un número de revisión independiente. Gracias a este sistema en el que almacenamos en la red nuestro trabajo, podemos acceder a él y descargarlo para trabajar desde diferentes máquinas.

4.4.2. Tecnologías y lenguajes de programación

Para el desarrollo de este proyecto hemos utilizado diferentes tecnologías como CEP para el procesamiento de eventos complejos, como motor de CEP hemos utilizado Esper y para diseñar los diferentes patrones de eventos el lenguaje de programación EPL.

También hemos utilizado un ESB para la integración de las diferentes tecnologías y conseguir trabajar con una arquitectura SOA 2.0 concretamente hemos trabajado con Mule ESB.

Para realizar todo el procesamiento de las alertas recibidas por *Snort*, el filtrado de información y posteriormente la comunicación con el responsable de seguridad vía email, se ha utilizado el lenguaje de programación Java.

4.4.3. Herramientas

Principalmente nos hemos valido de dos herramientas para el desarrollo del proyecto. Por una parte de un *sniffer* para la monitorización del tráfico de red y la generación de alertas, concretamente *Snort*.

La otra herramienta que hemos utilizado ha sido MuleStudio, que es el IDE sobre el que hemos realizado toda la programación. Está basado en el entorno de desarrollo Eclipse e incluye aspectos gráficos para el uso de las herramientas de Mule ESB.

4.4.4. Sistemas operativos y hardware

El proyecto desarrollado es multiplataforma ya se dispone de versiones de *Snort* y de Mule ESB tanto para plataformas Windows como UNIX. Además de toda la programación realizada con Java que no tiene problemas para ejecutarse en un sistema operativo u otro.

El software que hemos creado ha sido desarrollado en GNU/Linux, concretamente en la distribución 10.4 de Ubuntu.

En cuanto al hardware, el principal requisito es la conexión a internet. La máquina donde se ejecute esta aplicación debe poseer una conexión a internet bien por cable o bien por medios inalámbricos. El sistema no consume muchos recursos por lo tanto no se establecen ningún requisito especial que sea necesario en la máquina donde vaya a estar esta aplicación.

*

Capítulo 5

Desarrollo del proyecto

En este capítulo vamos a hablar del modelo de ciclo de vida que hemos decidido utilizar, los requisitos que tiene nuestro sistema y las fases de análisis, diseño, implementación y pruebas.

5.1. Modelo de ciclo de vida

Como hemos visto en el capítulo 3 hemos utilizado el modelo de ciclo de vida incremental. En este modelo se van realizando distintos incrementos hasta obtener el producto final. Gracias a este modelo podemos ir obteniendo versiones previas a la final a lo largo de todo el periodo de desarrollo.

En la figura 5.1 se puede observar mejor su funcionamiento, cada vez que acaba un ciclo se obtiene una versión incrementada del software y se aprecia como algunas etapas de un ciclo pueden solaparse en la ventana temporal de la siguiente iteración.

Este enfoque resulta muy útil ya que al ser evolutivo su uso es recomendado en proyectos en los cuales no están totalmente definidos al comienzo o pueden sufrir cambios a lo largo del desarrollo. También es útil en casos en los que hay distintos módulos independientes entre sí y que se pueden ir desarrollando en diferentes etapas.

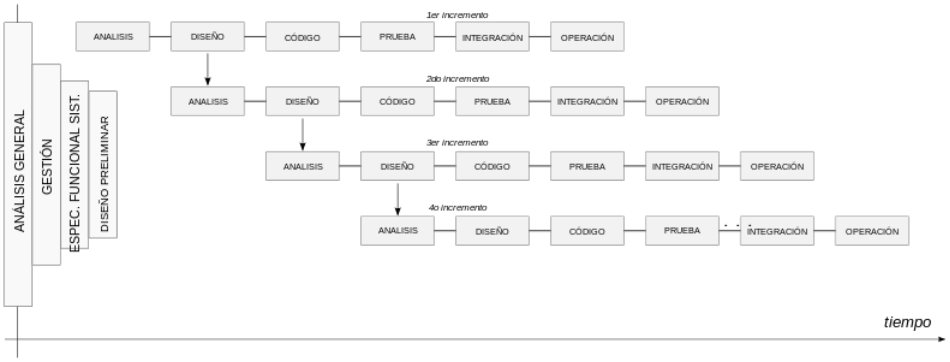


Figura 5.1: Modelo de ciclo de vida incremental

Un punto importante a la hora de desarrollar este proyecto utilizando este enfoque es la posibilidad de obtener versiones funcionales a lo largo de la producción, permitiéndonos así ver su funcionamiento y comprobar que puntos debíamos reforzar o incrementar.

5.2. Herramienta de modelado utilizada

Para realizar los diagramas UML que vamos a presentar en este documento hemos utilizado la herramienta multiplataforma gratuita llamada *yUML* con la que se puede trabajar de manera gratuita online en [35]

Esta herramienta nos permite realizar diagramas de casos de uso, de clases y de actividad. Para trabajar con esta aplicación, escribiremos la descripción de los diagramas mediante texto utilizando una serie de códigos y el software generará los diferentes diagramas.

5.3. Requisitos

5.3.1. Requisitos funcionales

En esta sección vamos a enumerar los requisitos funcionales que debe cumplir nuestro sistema.

- **Monitorizar el tráfico de red**. Nuestro proyecto debe tener la capacidad de estar continuamente monitorizando el tráfico de red a la espera de cualquier amenaza o riesgo para la integridad de la información contenida en nuestro equipo.
- **Generar alertas**. La aplicación debe generar una alerta cada vez que encuentre una situación que pueda poner en riesgo a nuestro sistema.
- **Transformación de alertas a eventos**. La aplicación debe ser capaz de procesar y transformar cada una de las alertas y convertirlas en un evento que se enviará al flujo de eventos.
- **Discriminación de la información**. El software tiene que extraer la información necesaria de las alertas para que el evento generado contenga sólo los datos que sean útiles para detectar ataques.
- **Filtrado de eventos**. El proyecto debe ser capaz de filtrar todos los eventos que se mueven por el flujo de eventos y quedarse sólo con los del tipo con el que estamos trabajando.
- **Detección de ataques**. Nuestra aplicación tiene que ser capaz de detectar a partir de los eventos seleccionados los ataques que se realicen a nuestro sistema y de los cuales tengamos definidos los patrones de eventos. Los ataques que el sistema va a detectar los podemos ver en la sección 5.5.5
- **Envío de la información**. La aplicación debe tener la capacidad de enviar un email con la información relativa al ataque al responsable de la seguridad del sistema.

5.3.2. Requisitos de información

Vamos a ver qué requisitos de información cumple nuestra aplicación.

- El sistema debe almacenar la información relativa a los eventos que se producen en el sistema.
- La aplicación debe ser capaz de almacenar la información relativa a los ataques que recibe el sistema.

5.3.3. Reglas de negocio

En esta sección vamos a ver las reglas de negocio que debemos tener en cuenta a la hora de desarrollar el software.

- La información que se genera relativa a los ataques que sufre el sistema debe ser enviada vía email al responsable de la seguridad del sistema.

5.3.4. Requisitos de interfaz

- El software debe ser capaz de integrarse con *Snort* para la monitorización del tráfico de red y generación de las alertas.
- La aplicación debe integrarse con Esper que es el motor de CEP con el que vamos a realizar el procesamiento de eventos complejos.

5.3.5. Requisitos no funcionales

Vamos a ver los requisitos no funcionales que debe cumplir nuestro sistema.

- **Tiempo de respuesta**. El sistema debe ser capaz de trabajar en tiempo real tanto para detectar las alertas como para identificar los ataques y responder a ellos.
- **Rendimiento**. El sistema debe tener un buen rendimiento sin consumir muchos recursos ya que debe ser una herramienta que se encuentre funcionando mientras se está trabajando con otras aplicaciones y no debe interferir ni reducir el rendimiento de estas otras.
- **Fiabilidad**. Este software debe tener una fiabilidad total al trabajar en un campo crítico como es la seguridad. El número de ataques que se produzcan y no detecte debe ser cero, ya que sólo con un ataque que no sea detectado puede ser suficiente para afectar seriamente a la integridad del equipo y de la información que contiene.
- **Mantenibilidad**. La aplicación debe ser sencilla de mantener, sobre todo el diseño de los patrones de eventos, ya que en cualquier momento se deben de poder integrar nuevos patrones que detecten otros ataques que no estén integrados aun en el sistema.

5.4. Análisis del sistema

5.4.1. Casos de uso

Un caso de uso es una descripción de las diferentes actividades que se deben realizar para realizar un proceso. Los actores son las entidades que participan en estas actividades. Gracias a estos diagramas podemos ver de manera más clara las comunicaciones y comportamientos de los sistemas y su interacción con los usuarios.

En la figura 5.2 podemos ver el diagrama de los casos de uso y sus relaciones.

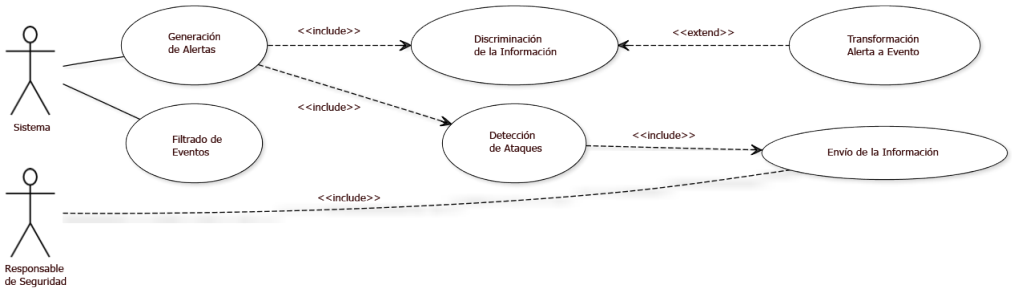


Figura 5.2: Diagrama de casos de uso

Caso de uso: Monitorizar tráfico de red

Descripción La aplicación se encuentra alerta monitorizando todo el tráfico que se produce en la red donde se encuentra el equipo.

Actor principal El sistema que monitoriza el tráfico de red.

Precondiciones La aplicación está en ejecución.

Postcondiciones El tráfico es monitorizado y el sistema está pendiente de las posibles alertas.

Escenario principal

1. Un usuario ejecuta el programa o se carga con el sistema operativo.

2. El sistema a través de su herramienta *sniffer* monitoriza el tráfico de red.

Escenario alternativo

- Un atacante realiza un ataque

    1. El sistema genera una alerta.

Caso de uso: Generación de alertas

Descripción Mientras el sistema está en ejecución y se encuentra monitorizando el tráfico de red, se reconoce una actividad potencialmente peligrosa en la red y el sistema genera una alerta referente a ese posible ataque.

Actor principal El sistema que genera la alerta relativo al posible ataque recibido.

Precondiciones El sistema está monitorizando el tráfico de red y se produce una actividad sospechosa en dicha red.

Postcondiciones El sistema genera un fichero de alerta relativo al posible ataque que se ha producido.

Escenario principal

1. Se produce una actividad sospechosa en la red.
2. El sistema detecta una situación potencialmente peligrosa.
3. El sistema genera un fichero de alerta con los datos relativos a esa actividad.
4. El sistema coloca ese fichero en la estructura de carpetas para ser procesado.

Caso de uso: Discriminación de la información

Descripción El sistema lee un fichero de alerta que ha sido generado tras la detección de un posible ataque y selecciona la información necesaria para la identificación de los ataques mediante el procesamiento de eventos complejos.

Actor principal El sistema que discrimina la información necesaria para la detección de ataques.

Precondiciones Se ha generado una alerta referente a un posible ataque.

Postcondiciones Se discrimina la información innecesaria y se prepara para la conversión del mensaje.

Escenario principal

1. El sistema comprueba si se han producido nuevas alertas.
2. El sistema lee el fichero con la alerta.
3. El sistema elimina la información innecesaria.

Caso de uso: Transformación de alertas a eventos

Descripción Se transforma el mensaje con la alerta después de ser filtrada su información y se genera un evento que se incluirá en el flujo de eventos para ser enviada al motor de CEP.

Actor principal El sistema que transforma el mensaje.

Precondiciones Se ha producido una actividad sospechosa y el sistema ha generado una alerta y ha procesado su información para eliminar los datos innecesarios.

Postcondiciones Se obtiene un evento que se incluye en el flujo de eventos.

Escenario principal

1. La información filtrada se transforma a un evento.
2. El sistema inserta ese evento en el flujo de eventos.

Caso de uso: Filtrado de eventos

Descripción Hay un flujo de eventos que se transmiten hacia el motor de CEP y se filtran aquellos que no son del tipo de las alertas de seguridad.

Actor principal El sistema que filtra los eventos según su tipo.

Precondiciones Existe un flujo de eventos y llega a este punto al menos un evento.

Postcondiciones Se filtran los eventos y pasan al motor de CEP aquellos que sean del tipo de las alertas de seguridad.

Escenario principal

1. Se observa el tipo del evento que ha llegado por el flujo de eventos.
2. Se comprueba que el evento es del mismo tipo que las alertas de seguridad.
3. El evento se envía al motor de CEP.

Escenario alternativo

- Se comprueba que el evento no es del mismo tipo que las alertas de seguridad
    1. El evento se desestima y se compara el siguiente.

Caso de uso: Detección de los ataques

Descripción Al menos un evento ha llegado al motor de CEP donde el sistema comprueba si concuerda con algún patrón de evento relativo a los ataques de los que se dispone.

Actor principal El sistema que procesesa los eventos en busca de un ataque producido.

Precondiciones Al menos un evento llega al motor de CEP para su procesamiento.

Postcondiciones Se detecta un ataque que se ha producido en el sistema y se envía dicha información al *listener* correspondiente.

Escenario principal

1. Un evento llega al motor de CEP.
2. El sistema encuentra un patrón que se identifica con el evento.
3. El sistema envía la información del ataque detectado al *listener* correspondiente.

Escenario alternativo

- Ningún patrón concuerda con el evento
  1. Se almacena la información relativa al evento recibido.
  2. Se procesa el siguiente evento

Caso de uso: Envío de la información

Descripción El sistema envía la información relativa a un ataque que se ha producido al responsable de la seguridad del sistema.

Actor principal El sistema que envía la información mediante un email.

Actor secundario El responsable de la seguridad informática que recibe la información y actúa en consecuencia.

Precondiciones Se ha detectado un ataque mediante el procesamiento de eventos complejos y se ha informado al *listener* correspondiente.

Postcondiciones Se envía un email al responsable de la seguridad del sistema con toda la información relativa al ataque detectado.

Escenario principal

1. El sistema genera un informe con la información relativa al ataque recibido.
2. El sistema envía dicha información mediante un email al responsable de la seguridad del sistema.
3. El responsable de la seguridad actúa respecto al ataque del que ha sido informado.

5.4.2. Modelo conceptual de datos del dominio

En esta aplicación las entidades con las que vamos a trabajar son las alertas y los ataques. De ambas vamos a almacenar la misma información que será la que utilicemos para identificar los patrones de eventos y detectar así cuando estamos ante un ataque a nuestro sistema. Esta información sera la que utilizaremos para generar un informe que se mostrará por pantalla y además será enviado mediante un email al responsable de la seguridad del sistema.

La información que almacenamos de cada una de estas entidades es la siguiente:

- *Timestamp.* Es el momento en el que se produce la alerta.
- Ip de origen. Dirección ip del atacante.
- Puerto de origen. Puerto utilizado por el atacante.
- Ip de destino. Dirección ip de la máquina que ha sido atacada.
- Puerto de destino. Puerto que ha sido atacado en la máquina de la víctima.

Con esta información como podremos ver posteriormente podremos detectar cualquier amenaza comparándola con los patrones que tenemos definidos en el sistema. Es muy importante señalar que para el desarrollo de este proyecto no ha sido necesario el uso de una base de datos. Esto es principalmente debido a que no vamos a realizar consultas sobre una base de datos en ningún momento sino que *Snort* genera los ficheros de alerta y los incluye en una carpeta reservada para ello. Posteriormente las alarmas que generemos sobre cada uno de los ataques van a ser almacenadas en la cuenta de correo electrónico del responsable de seguridad que es el encargado de administrar estas alarmas.

Además, el uso de una base de datos ralentizaría la respuesta de la aplicación y en este caso estamos diseñando un sistema que debe de actuar en tiempo real con lo que debemos priorizar el tiempo de respuesta.

5.4.3. Diagramas de secuencia

En esta sección vamos a mostrar el diseño de los diagramas de secuencia referentes a los casos de uso con los cuales podemos ver mejor la secuencia de eventos que se producen en el sistema. Gracias a esto vamos a poder identificar mejor las operaciones que se producen.

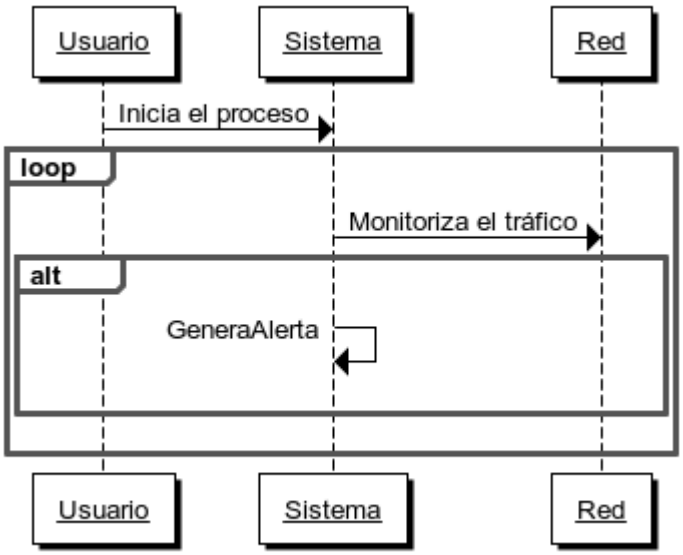


Figura 5.3: Diagrama de secuencia de monitorizar el tráfico de red

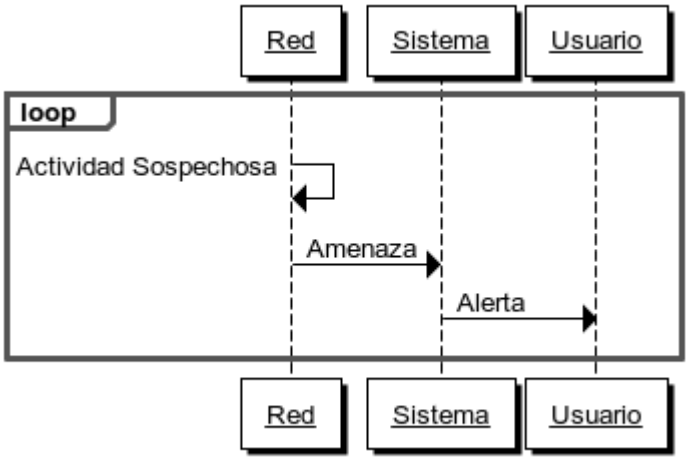


Figura 5.4: Diagrama de secuencia de generación de alertas

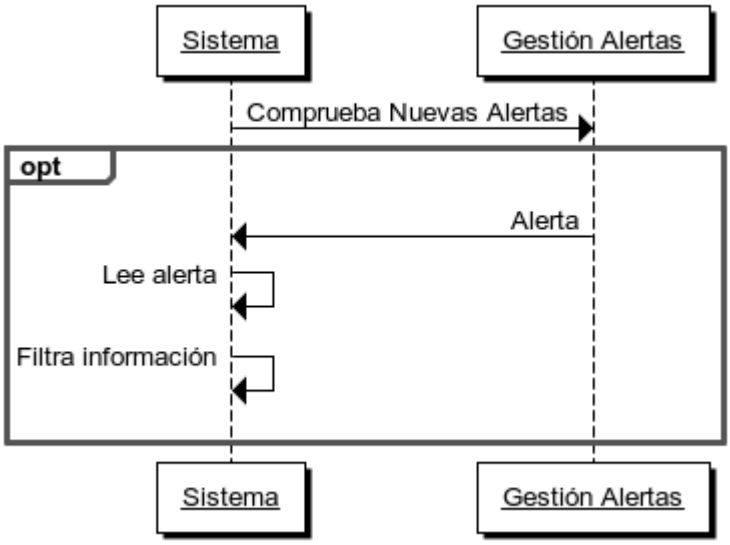


Figura 5.5: Diagrama de secuencia de discriminación de la información

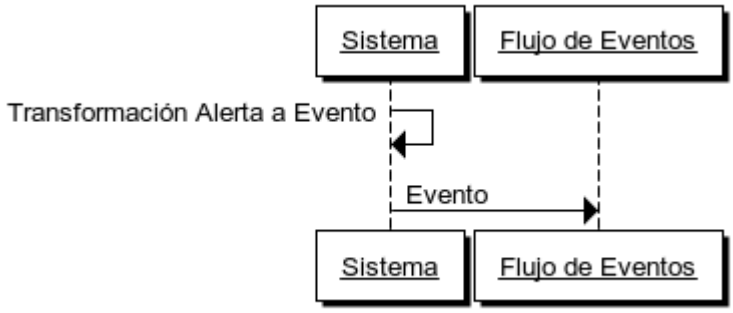


Figura 5.6: Diagrama de secuencia de transformación de alerta a eventos

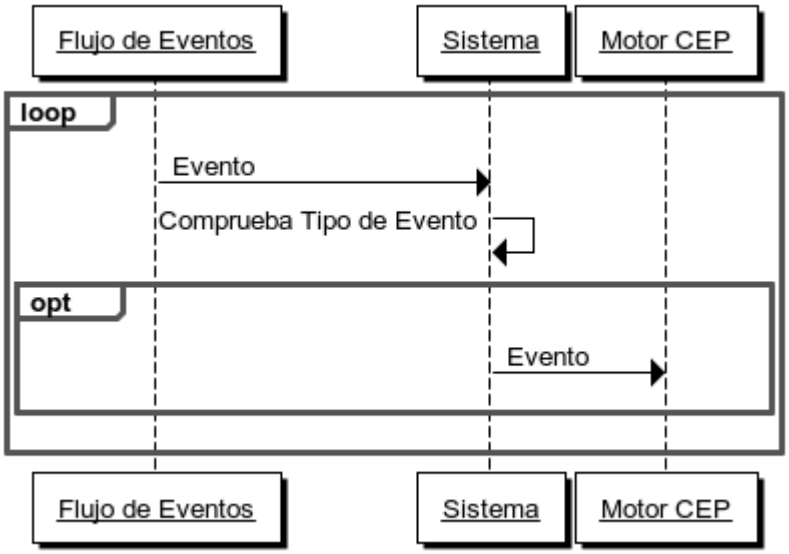


Figura 5.7: Diagrama de secuencia de filtrado de eventos

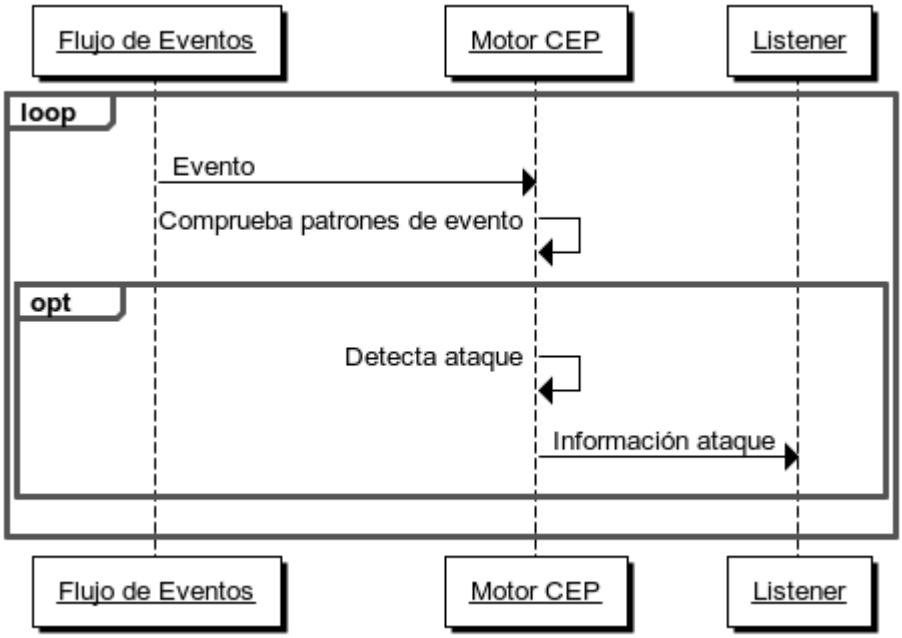


Figura 5.8: Diagrama de secuencia de detección de los ataques

Figura 5.9: Diagrama de secuencia de envío de la información

5.5. Diseño del sistema

Como hemos visto hasta ahora, el objetivo final del proyecto es crear una aplicación que sirva para aumentar la seguridad de los equipos en el campo de la detección de las intrusiones a través de la red.

En esta sección vamos a ver qué es y cómo trabaja nuestra aplicación, además entraremos a valorar cada uno de los ataques que nuestro sistema es capaz de reconocer y cuál ha sido la estructura que hemos elegido para integrar las diferentes tecnologías.

5.5.1. La aplicación

El producto final que hemos desarrollado en este proyecto es una herramienta que tiene la capacidad de trabajar junto a un *sniffer* y que aprovechando su capacidad de monitorización de red y generación de alertas, podemos a partir de ahí y utilizando el procesamiento de eventos complejos, detectar ataques que pongan en riesgo la integridad de la información contenida en el equipo.

La aplicación ha sido diseñada para detectar ocho posibles ataques, que enumeramos a continuación, aunque el sistema está definido de manera que estos ataques puedan ser ampliados sin problemas. Actualmente los patrones de eventos que se encuentran disponibles son los siguientes:

- Ataque de denegación de servicio.
- Ataque distribuido de denegación de servicio.
- Ataque *smurf*.
- Ataque *Land*.
- Ataque *supernuke*.
- Escaneo de puertos.
- Ataque de tipo *flood* al email.
- Ataque al FTP.

En la sección 5.5.5 veremos en profundidad cada ataque, cuáles son sus características, cómo se realiza y cómo podemos detectarlos. En esta misma sección vamos a ver la estructura de los patrones que hemos diseñado para detectar estos ataques.

Para comprender mejor el diseño del software que hemos desarrollado, en la siguiente sección vamos a ver su estructura, donde podremos observar cómo se ha conseguido desarrollar una arquitectura a partir de los enfoques que hemos visto en los capítulos anteriores.

5.5.2. Estructura de la aplicación

Hasta ahora hemos hablado mucho de las diferentes arquitecturas que existen, hemos visto como las arquitecturas SOA están dirigidas a servicios y son ampliamente utilizadas pero no están recomendadas para trabajar en tiempo real. También hemos visto que las arquitecturas EDA son las dirigidas por eventos y se asemejan a nuestro estilo de trabajo y que una posible solución era la unión de estas dos arquitecturas a través de un ESB y utilizando CEP para obtener una arquitectura SOA 2.0 Pero hasta ahora no hemos visto realmente cómo hemos diseñado nuestro sistema a partir de este enfoque.

En la figura 5.10 podemos ver la estructura de nuestra aplicación. Podemos distinguir tres partes bien diferenciadas. Por una parte, el productor de eventos, que sería el sensor, en este caso un *sniffer* que es *Snort*. La segunda parte que es el ESB donde se realizan las transformaciones y la tercera parte, después de pasar por el motor CEP, el suscriptor a la información que generamos tras la detección de los ataques.

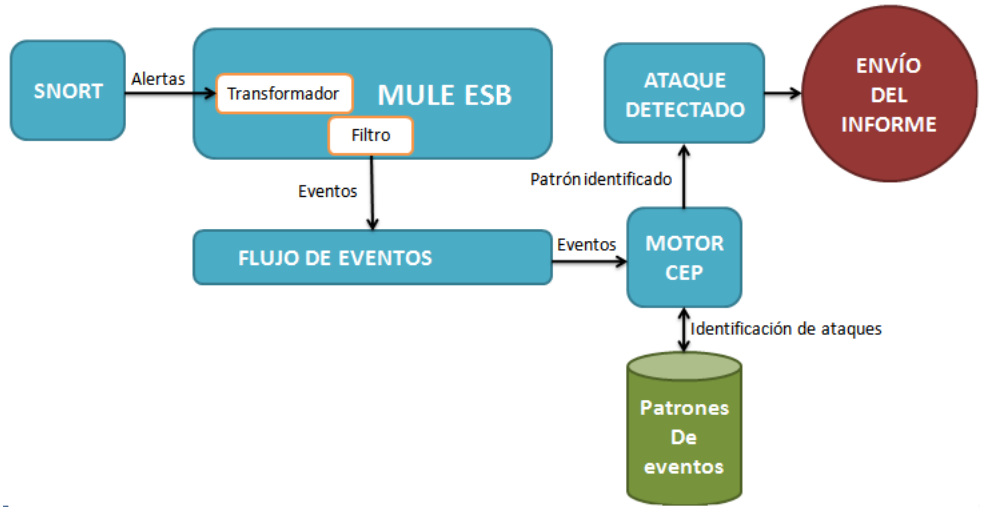


Figura 5.10: Estructura de la aplicación

A partir de esta figura podemos entender mucho mejor el diseño y funcionamiento de nuestra aplicación. En primer lugar tenemos el productor de eventos. Hemos visto que en las arquitecturas SOA tenemos un productor que puede ser un servicio web, un sensor o una aplicación externa. En este caso estamos utilizando un *sniffer*, *Snort*, que es el encargado de la monitorización del tráfico de red, y va a generar una alerta cada vez que detecte una situación anómala en la red y que pueda ser peligrosa para nuestro sistema.

Uno de los puntos más importantes es ver cómo podemos integrar *Snort* en nuestro sistema, en la sección 5.5.4 vamos a ver cómo hemos podido hacer esto a través del ESB.

Este ESB es la pieza fundamental donde se sustenta toda la arquitectura del sistema. Por una parte es capaz de conseguir la integración de *Snort*, pero no sólo eso, también es capaz de obtener las alertas que genera esta herramienta y mediante transformaciones que hemos diseñado y posteriormente veremos, transformarlas a un formato adecuado y conteniendo tan sólo la información necesaria para el procesamiento y detección de ataques.

Gracias a esta herramienta vamos a obtener un evento relativo a una alerta detectada y que estará compuesto por la información relativa al tiempo, puerto de origen, puerto de destino, ip de origen e ip de destino. Este evento se introduce en el flujo de eventos que está continuamente transportando los eventos que se suceden en el sistema con el fin de que el motor de CEP detecte los ataques. Pero antes de eso, gracias al ESB vamos a incluir un filtro en este flujo de eventos donde podremos observar el tipo de los eventos que se mueven por el flujo y descartar aquellos que no sean del tipo que queremos examinar e identificar.

Los eventos que han conseguido pasar ese filtro irán al motor de CEP. En este motor se encuentran todos los patrones de eventos que hemos diseñado relativos a los ataques contra la seguridad que hemos tenido en cuenta. Estos patrones están diseñados en EPL que es el lenguaje que utiliza Esper, el motor de CEP que hemos decidido utilizar. Mediante el procesamiento de eventos complejos a medida que lleguen los eventos por el flujo al motor, éste va a compararlos con los patrones que se encuentran en el sistema para ver si se detecta algún ataque. En caso de que no sea así, se trata el siguiente evento, comprobando así la relación entre eventos y utilizando varios de los que van llegando para la detección de los ataques.

Cuando a partir de la llegada de una serie de eventos relativos a un mismo ataque, éste sea detectado por el motor de CEP tras la identificación de un patrón de evento que se asemeja a los eventos que han ido llegando a través del flujo, la información relativa al ataque será enviada al *listener* correspondiente. En ese punto se genera un informe que incluye qué ataque se ha producido en el sistema, en qué momento, quién ha sido el atacante y cuáles han sido los puertos y la ip que ha sido atacada.

Este *listener* va a mostrar la información por pantalla, además de enviar ese informe mediante un email al suscriptor de la información, es decir, siguiente la estructura de una arquitectura SOA 2.0, al servicio consumidor de la información. En este caso esta información será enviada al responsable de la seguridad del sistema, que recibirá en tiempo real los datos del ataque que se ha producido y será el encargado de actuar en relación al ataque.

En este punto, se podía haber optado por realizar alguna acción automática para contrarrestar el ataque, pero esta opción queda descartada ya que creemos que es mejor que sea el responsable de la seguridad el encargado de tomar la decisión de que acción debe llevar a cabo. Esto es en parte debido a que hay escenarios en los que ciertos ataques no van a causar ningún daño al equipo, y la información que va a llegar al responsable va a ser meramente informativa, en cambio, si realizáramos alguna acción como cerrar ciertos puertos, si estamos tratando la seguridad en una empresa, podemos haber realizado una sobre protección, impidiendo que algún proceso de negocio se pueda llevar a cabo cuando realmente el ataque no pone en riesgo en ningún momento la seguridad de la información contenida en el equipo.

5.5.3. Enmarcación de este proyecto dentro de los sistemas de seguridad

Como ya hemos visto, actualmente el número de herramientas para aumentar el nivel de seguridad en los sistemas informáticos es muy grande y crece cada día. Dentro de estas herramientas existen diferentes tipos, cada una de ellas tienen un objetivo concreto dentro del campo de la seguridad. Principalmente podemos diferenciar los siguientes campos.

- Seguridad física y acceso a los equipos.
- Seguridad en la red y en las comunicaciones.
- Cifrado y encriptación de datos.
- Planificación y administración.
- Defensa contra los virus, gusanos y *malware*.
- Control de acceso a los sistemas.
- Monitorización de eventos.
- Estrategias y políticas de seguridad.

Nuestro sistema se enmarca dentro del campo de la seguridad en la red, más concretamente en el control de acceso a los sistemas a través de la red controlando la intrusión mediante ataques al sistema y de la monitorización de eventos, exactamente de los eventos relativos a las alertas de seguridad que se producen en la red.

Dentro de las herramientas conocidas nuestro sistema se asemeja más a un IDS, ya que es un sistema preventivo para la detección de intrusiones y de amenazas de seguridad. Dentro de los IDS lo podríamos identificar como un NIDS, ya que está basado en red.

Como hemos comentado anteriormente, el uso de una herramienta como la que hemos desarrollado no está indicada para ser utilizada como única protección, sino que el sistema para estar dotado de una buena protección debería unir el uso de varias herramientas destinadas a la seguridad, cada una con la misión de proteger un aspecto determinado de los sistemas informáticos.

5.5.4. Diseño de la estructura en el ESB

Como hemos visto, la estructura SOA 2.0 se soporta sobre un ESB, concretamente vamos a utilizar Mule ESB. Ya hemos visto que este ESB permite trabajar de manera ligera con diferentes arquitecturas, ofreciendo las herramientas necesarias para integrar diferentes aplicaciones y tecnologías. En el anexo correspondiente veremos todas las opciones que nos ofrece Mule para desarrollar aplicaciones.

Figura 5.11: Diseño de la estructura en Mule ESB

En la figura 5.5.4 podemos ver la estructura que hemos diseñado sobre Mule para la implementación de este proyecto. Hemos utilizado un IDE visual

como es MuleStudio, que nos permite utilizar y configurar las diferentes herramientas de una manera gráfica, mejorando la experiencia del usuario y evitando así tener que trabajar con ficheros de configuración XML.

Como podemos ver en la figura 5.11 tenemos 5 elementos diferenciados, vamos a explicar cual es la función de cada uno sin entrar en su implementación, que la veremos en la sección 5.6 de este documento.

En primer lugar tenemos un *endpoint* de tipo *File*, gracias a esto vamos a poder estar vigilando constantemente la carpeta donde se van a almacenar las alertas de *Snort*. Como hemos visto, cada vez que se produce una actividad sospechosa en la red, *Snort* genera un fichero de alerta y lo coloca en una carpeta específicamente reservada para ello. Con este *endpoint* lo que vamos a lograr es estar en permanente estado de alerta y cada vez que haya un nuevo fichero en esa carpeta, Mule lo va a consumir. Esta herramienta tiene una doble funcionalidad en el momento en el que obtiene este fichero. Por una parte realiza una copia de él y lo envía a una carpeta de *backup* asegurándonos así el no perder la información. Por otra parte, envía el fichero original al siguiente componente de Mule ESB que sería un transformador. El fichero una vez consumido desaparece de la carpeta de *Snort* y el ESB se mantiene a la espera de que un nuevo fichero sea generado.

El segundo elemento que podemos observar en el flujo de Mule ESB es un transformador, concretamente uno del tipo *File-to-String*, es decir, un componente que recoge un fichero y lo transforma a una cadena. El objetivo de realizar esta transformación es que para poder leer un fichero necesitamos convertirlo a un formato que nos permita trabajar con él, por lo tanto antes de leer esta alerta y procesar la alerta, tenemos que realizar esta transformación. Una vez que hayamos convertido la alerta de tipo fichero a una alerta de tipo cadena de caracteres, vamos a enviarla al siguiente componente del flujo.

Este tercer componente que vamos a utilizar, al igual que antes, es de tipo transformador. En este caso es un transformador *String-to-Event*. Lo que recibe este elemento es la alerta que generó *Snort* pero en formato de cadena de caracteres y en este punto se realiza una doble función. Por una parte

se realiza la operación de discriminación de la información. Como hemos visto, *Snort* genera un fichero de alerta con mucha información pero para la detección de los ataques no es necesario conocer todos los detalles de la alerta, por lo tanto tenemos que descartar aquellos que no nos van a ayudar a identificar los ataques. Mediante un programa que hemos diseñado en Java y que veremos su implementación en el próximo capítulo, leemos este fichero de alerta y nos quedamos únicamente con su *timestamp*, su puerto de origen, puerto de destino, ip de origen e ip de destino. Una vez que tenemos únicamente esta información referente a la alerta entra en juego la segunda funcionalidad que implementamos en este componente que no es otra que la propia transformación. Hemos visto que a la hora de detectar las amenazas vamos a realizarlo mediante el motor de CEP, y esto lo que lee son los eventos que recibe a través de un flujo de eventos. Pero en este flujo, los únicos elementos que pueden navegar en él son eventos, y para ello se realiza esta transformación, creando un evento solamente con la información que hemos filtrado en este mismo componente. Una vez que tenemos creado el evento y antes de incluirlo en el flujo de eventos debe pasar por otra herramienta que nos proporciona el ESB.

El cuarto punto de este flujo de Mule ESB es un *payload*, es decir, un filtro que vamos a establecer para comprobar el tipo de los eventos que navegan por el flujo. El objetivo final es que las alertas que se obtuvieron en un principio y posteriormente se transformaron a un evento lleguen al motor de CEP para su procesamiento, pero no podemos permitir que a este motor lleguen eventos de un tipo que nada tenga que ver con la misión del software que no es otra que reconocer ataques de seguridad. Por lo tanto con este filtro lo que vamos a hacer es ver el tipo de cada uno de los eventos que pasen por este componente y desechar aquellos que no cumplan la condición que hemos indicado. Los que sí la cumplan, es decir, aquellos eventos que sean relativos a una alerta de seguridad pasarán al último paso del flujo de Mule ESB.

El último paso es un componente de Java, a este elemento sólo han llegado las alertas en formato de evento que tienen la información necesaria para detectar ataques de seguridad. La funcionalidad de este componente es integrar Mule ESB con el motor de CEP, para ello se ha diseñado un programa en Java que envía estos eventos a dicho motor, estableciendo así una comunicación en tiempo real entre el generador de eventos *Snort* y el motor Esper donde se

va a realizar el procesamiento de los eventos complejos con el fin de detectar ataques al sistema.

Hemos visto los componentes de Mule ESB que nos permiten realizar las integraciones de *Snort* con el ESB y de éste con el motor de CEP. En el próximo capítulo veremos como hemos implementado esto. En nuestro proyecto no sólo vamos a usar *Snort* y el ESB sino que tenemos que programar todo lo relativo al envío de información al responsable de la seguridad y previamente a la identificación de los ataques identificando los diferentes patrones de eventos, para ello en la próxima sección vamos a ver cuales son estos ataques que vamos a tener en consideración y posteriormente veremos como los podemos describir de manera que podamos diseñarlos mediante el lenguaje EPL de Esper.

5.5.5. Patrones CEP aplicados a la seguridad

En el campo de la seguridad informática el número de ataques existente es muy amplio, nosotros nos vamos a centrar en los ataques o intrusiones que se pueden producir a través de la red, y dentro de estos ataques, vamos a tratar concretamente ocho de los más usuales. Vamos a diseñar patrones de eventos para los ataques DOS, DDOS, *Smurf*, *Land*, *Supernuke*, TCP *Connect*, *flood* al email y ataque al puerto de FTP. Por supuesto el sistema está diseñado de tal manera que añadir nuevos patrones resulte sencillo para el desarrollador. Los ataques a la seguridad pueden ser catalogados como ataques activos o pasivos. Los activos son aquellos que alteran o realizan alguna modificación sobre el flujo de datos, mientra que los ataques pasivos son aquellos que tienen como objetivo la obtención de datos, la escucha, monitorización o interceptación de datos. Como podemos observar los ataques que vamos a detectar si bien tienen como objetivo obtener la información almacenada en nuestro sistemas, muchos de ellos se pueden enmarcar en los ataques activos ya que usan técnicas como *spoofing* para infiltrarse en la red o intentan modificar ciertas características de nuestro sistema para poder acceder a él aprovechando las vulnerabilidades generadas.

Vamos a tratar de explicar cada uno de estos ataques para que veamos más claro cómo se producen, con que intención y cuál es la manera de detectarlos.

También vamos a definir los patrones que nos permiten detectar los ataques, para ello vamos explicar que en que situaciones vamos a reconocer estos ataques y que atributos necesitamos conocer de ellos. En esta sección no vamos a hablar de la implementación de estos patrones ya que para ello vamos a tener una parte exclusiva en la sección de este documento dedicada a la implementación de los diferentes elementos del proyecto.

5.5.5.1. Ataque DOS

Estas son las siglas de *Denial of Service,* es decir, es un ataque de denegación de servicio y el objetivo es conseguir que un recurso o servicio de la máquina atacada quede inaccesible gracias a la sobrecarga de consumo de recursos en la máquina atacada o del consumo de su ancho de banda.

La manera de realizar este ataque es saturar un puerto de la máquina objetivo enviando muchas peticiones a ese mismo puerto, consiguiendo así la sobrecarga del servicio. En el momento en el que la máquina atacada no puede responder se llama "denegación" ya que no es capaz de ofrecer respuesta a todas las peticiones que le llegan.

Para realizar este tipo de ataques siempre se utiliza el protocolo TCP/IP y el atacante puede tener diferentes objetivos finales como puede ser consumir los recursos de la máquina, ya sea el ancho de banda, capacidad de memoria, tiempo del procesador... modificar la información de la configuración, las rutas de encaminamiento o la información del estado interrumpiendo sesiones TCP mediante un TCP *reset*. Otros objetivos pueden ser la interrupción de los componentes físicos de red, la obstrucción de la comunicación entre el usuario y el servicio atacado o principalmente conseguir el acceso a la máquina aprovechando algún servicio que se ha conseguido tirar.

Este tipo de ataques, una vez que han logrado el objetivo de saturar un puerto o recurso, es prácticamente imposible solucionarlo a corto plazo, por lo tanto la mejor manera de de evitarnos problemas es la prevención, y en el momento en el que haya indicios de este tipo de ataques, aplicar soluciones como pueden ser bloquear una ip entrante o cerrar algún puerto afectado.

El patrón *DOSAttackPattern* nos va a permitir detectar ataques de denegación de servicio. Vamos a estar ante una situación de ataque DOS cuando en el periodo de un minuto se produzcan 10 eventos que cumplan las siguientes características:

- La dirección ip de origen es la misma en todos los eventos.
- La dirección ip de destino es la misma en todos los eventos.
- El puerto atacado es el mismo en todos los eventos.

Con estos datos podemos reconocer que estamos ante una misma máquina que ha enviado en el tiempo máximo de un minuto 10 paquetes sobre un mismo puerto de nuestra máquina, con lo que tenemos tiempo suficiente para actuar antes de que se produzca un mal mayor.

Con estos 10 eventos simples que componen el evento complejo es difícil que el atacante pueda dañar nuestra estructura, y esto es precisamente lo que queremos conseguir, que el responsable de la seguridad pueda actuar ya sea bloqueando la ip de origen o cerrando el puerto de destino o tomando la decisión que crea conveniente.

Vamos a generar un informe con la dirección de ip del atacante y el puerto desde el que nos ha atacado. También incluiremos la dirección ip atacada y el puerto que ha sido objeto del ataque además del momento en el que se ha producido este ataque.

5.5.5.2. Ataque DDOS

Este ataque es similar al que acabamos de ver pero entra en juego un nuevo elemento y es que este ataque se realiza de manera distribuida, es decir, no es sólo un atacante, sino que son varios los que desde diferentes puntos intentan lograr los mismos objetivos que vimos en el ataque anterior. Gracias a que son varios los atacantes, la potencia del ataque aumenta bastante ya que se genera un flujo de información bastante importante desde diferentes puntos.

Para realizar estos ataques se suele utilizar una *botnet*, es decir, un conjunto de ordenadores que han sido infectados y actúan como *zombies*, por lo tanto puede ser un único atacante que en primer lugar ha obtenido el control de diferentes máquina y las utiliza para atacar un único objetivo. Puede que incluso los usuarios de los *hosts* zombies no sepan que su máquina está siendo utilizada para realizar un ataque contra un servidor.

Los objetivos de este tipo de ataque son los mismos que los que hemos visto en el ataque DOS con la diferencia de que ahora el ataque es más potente debido al número de atacantes que realizan peticiones en paralelo. También algunos administradores de seguridad han utilizado este ataque contra su propio sistema para comprobar la robustez de éste y ver que capacidad es capaz de resistir.

Como podemos ver en la figura 5.12, la estructura de un ataque DDOS suele ser siempre la misma, un atacante infecta diferentes máquinas y a partir de ahí se unen sus peticiones contra un mismo servidor objetivo. En otras ocasiones pueden ser varios atacantes que estén de acuerdo en realizar un ataque sobre un mismo sistema.

El patrón *DDOSAttackPattern* nos va a permitir detectar ataques distribuidos de denegación de servicios. Estaremos en una situación de ataque DDOS cuando en le periodo de un minuto se produzcan 10 eventos que cumplan las siguientes condiciones:

- La dirección ip de origen es distinta.
- La dirección ip de destino es la misma en todos los eventos.
- El puerto atacado es el mismo en todos los eventos.

Gracias a esta información podemos reconocer qué diferentes máquinas han enviado en el tiempo máximo de un minuto 10 paquetes sobre el mismo puerto de nuestra máquina. La diferencia respecto al patrón que identifica un ataque DOS es que en este caso los *hosts* que nos atacan son diferentes ya que son los distintos robots o máquinas *zombies* que están siendo utilizadas para lanzar el ataque sobre nosotros.

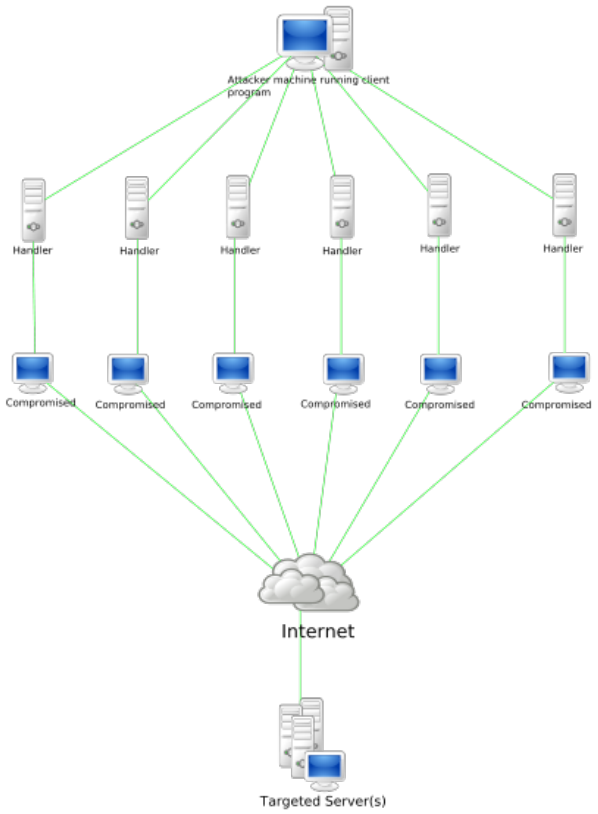


Figura 5.12: Estructura ataque DDOS

Con este patrón podemos detectar un ataque distribuido de denegación de servicio antes de que se produzcan daños importantes en nuestro sistema, consiguiendo que se pueda tomar una decisión como bloquear el rango de ips que nos atacan o cerrar el puerto atacado.

En el *listener* correspondiente se va a generar un informe con los datos necesarios para que el responsable de la seguridad sea el encargado de tomar las decisiones adecuadas en cada caso.

5.5.5.3. Ataque Smurf

Este tipo de ataques está basado en los ataques de denegación de servicio ya que lo que intenta es inundar un objetivo realizando un *ping* al *broadcast* pero utilizando la técnica de *spoofing*.

La estructura del ataque es normalmente un único atacante que realiza muchos *pings* sobre un mismo puerto a la dirección de *broadcast*. Normalmente el *router* que reciba esas peticiones las va a redirigir a los *hosts* que responderán a estos mensajes ICMP aumentando el tráfico en su subred. Si la red tiene múltiples acceso a *broadcast*, todas las máquinas van a responder a cada uno de los paquetes, por lo tanto esas respuestas irán hasta la ip atacada consiguiendo el atacante así su objetivo de inundación de peticiones.

La manera de detectar estos ataques es detectar la técnica de *spoofing* en la ip de origen, es decir, el atacante camuflará su dirección ip como si fuera la dirección ip de la máquina atacada para lograr así el objetivo de inundación, por lo tanto una de las maneras más efectivas de contrarrestar este ataque es la prevención en cuanto que se observe algún paquete sospechoso.

El patrón *SmurfAttackPattern* nos permite detectar ataques *smurf* que utilizan la técnica de *spoofing*. Vamos a identificar este ataque cuando se produzcan 2 eventos en un minuto que tengan las siguientes características:

- La dirección ip de origen es la misma que la dirección ip de destino.
- La dirección ip de destino es la misma en ambos eventos.
- El puerto de origen es distinto que el puerto de destino.

Con esta información podremos reconocer que un atacante está simulando que su dirección ip es la misma que la nuestra para conseguir que el sistema se inunde de peticiones. En este caso si vemos que se producen dos eventos similares podremos sospechar y lanzar la detección de este ataque para prevenir que el atacante consiga su objetivo.

Con este patrón vamos a detectar un ataque *smurf* antes de que se produzcan daños irreversibles y para ello vamos a generar un informe con los datos asociados a este ataque y será enviado al responsable de la seguridad para que tome las medidas adecuadas en este caso.

5.5.5.4. Ataque *Land*

Otra manera de colgar una máquina y aprovechar esta situación para acceder a él es realizar un ataque *land*. Estos ataques tienen la característica de que el atacante camufla sus datos como los del atacado, es decir, tanto la ip como el puerto del atacante van a parecer que son los mismos que los de la máquina atacada.

La forma en la que los atacantes realizan esto es la de enviar paquetes TCP/SYN falsificando las direcciones de ip y el puerto, gracias a esto la máquina atacada se va a responder continuamente a sí misma consiguiendo que finalmente falle.

Si bien el objetivo de este ataque en principio podría ser sobre cualquier puerto, los atacantes usan como objetivos el puerto 113 y el puerto 139. El puerto 113 es el puerto de *auth*, que suele ser utilizado en máquinas UNIX por el demonio *identd*. Es un puerto para la identificación de usuarios y se puede obtener información de los usuarios de la máquina a través de él. El puerto 139 es un puerto de *NetBIOS* usado para el servicio de sesiones, este puerto suele ser el objetivo de muchos atacantes en máquinas que operan bajo Windows, ya que este sistema operativo utiliza este puerto para la compartición de archivos y de impresoras en la red, haciendo de éste un objetivo vulnerable que los atacantes suelen intentar explotar.

El patrón *LandAttackPattern* nos permite detectar ataques *land* dónde el atacante se intente camuflar utilizando nuestros datos para conseguir atacar nuestro sistema. Vamos a detectar este ataque cuando en el periodo de un minuto encontremos dos eventos que cumplan las siguientes condiciones:

- La dirección ip de origen es idéntica a la dirección ip de destino.

- La dirección ip de destino es la misma en ambos eventos.
- El puerto de origen es igual al puerto de destino.
- El puerto de destino es el 113 o el 139.

Con estos datos podremos afirmar que alguien está intentando realizar un ataque *land* sobre nuestro sistema para intentar que la máquina objetivo se responda a sí misma y falle por inundación. El atacante va a realizar este ataque sobre el puerto 113 o sobre el puerto 139.

En cuanto que detectemos dos eventos iguales podemos lanzar la alerta al responsable de la seguridad del sistema para que pueda realizar las acciones preventivas que crea conveniente evitando así que se produzca un fallo en el sistema.

5.5.5.5. Ataque *Supernuke*

Este es otro ataque que se suele dar contra máquinas que trabajan en Windows. El objetivo es colgar a los equipos que escuchen a través del puerto TCP/UDP 137, 138 o 139. Los atacantes envían fragmentos de paquetes *Out Of Band* y la máquina atacada los detecta como inválidos y puede pasar a un estado inestable.

La manera de detectar estos ataques es escuchar los puertos de *NetBIOS* y ver si se produce alguna anomalía en los paquetes que recibe algunos de esos puertos. *NetBIOS* es una capa software para el acceso a los servicios de red. Gracias a esta utilidad se consigue comunicar a las aplicaciones con la red e identifica a cada máquina con un nombre único en la red. Cada una de estas máquinas se comunica estableciendo sesiones, usando datagramas o mediante *broadcast*. Todas las comunicaciones que se realicen de esta manera son del tipo punto a punto.

NetBIOS permite las comunicaciones que son orientadas a conexión (TCP) y las que son no orientadas a conexión (UDP). Cada vez que hay un programa

que necesita usar alguno de estos servicios de *NetBIOS* envía una interrupción de este tipo indicando que requiere un servicio de red. La interfaz de *NetBIOS* define que programas pueden usar este tipo de interrupciones.

Los puertos que se utilizan en este ataque y que son los que debemos identificar en los patrones de eventos son el puerto 137 que es el encargado de los servicios de nombres en *NetBIOS*, el puerto 138 que es el encargado de los servicios de datagramas y el puerto 139 que como vimos en el ataque anterior es el encargado de establecer las sesiones.

El patrón *SupernukeAttackPattern* nos permite detectar ataques *supernuke* en los cuáles uno o varios atacantes van a intentar hacer una denegación de servicio sobre un puerto de *NetBIOS*. Detectaremos este ataque cuándo en el periodo de un minuto observemos 10 eventos que tengan las siguientes características:

- La dirección ip de destino es la misma en todos los eventos.
- El puerto de destino es el 137 o el 138 o el 139.

Con esta información estamos teniendo en cuenta que puede ser un sólo atacante o varios y que los puertos atacados serán los dedicados a *NetBIOS* ya sean TCP o UDP 137, 138 ó 139.

Vamos a detectar este ataque en el momento en el que encontremos 10 eventos similares teniendo así el tiempo de maniobra suficiente para que el responsable de la seguridad pueda actuar correctamente evitando así un riesgo mayor.

5.5.5.6. Ataque Escaneo de puertos TCP

Uno de los ataques más conocidos y que más se realizan son los escaneos de puertos. Normalmente se utiliza un software para realizar esto y en muchas ocasiones, es el primer paso para realizar otro ataque posterior más potente. El objetivo es analizar el estado de todos los puertos (o algunos de ellos) en una máquina objetivo. Con estos escaneos se puede conocer si el puerto se encuentra abierto o cerrado e incluso si está protegido con algún sistema de cortafuegos.

Con este ataque se pueden extraer conclusiones como cuáles son las vulnerabilidades del sistema que estamos planteándonos atacar y que servicios ofrece esa máquina. Incluso podemos sacar datos sobre el sistema operativo con el que se está trabajando con el fin de obtener la mayor información posible para realizar nuestro posterior ataque. Un administrador de red también puede realizar esto sobre su propia red para conocer la seguridad que ofrece su equipo con respecto a posibles atacantes.

El formato de este ataque sería el siguiente. Un atacante envía un paquete *SYN* a la máquina de destino, ésta contesta con un paquete *SYN/ACK* y por último la máquina de origen contesta con un paquete *ACK*. Una vez realizado esto se ha establecido la conexión TCP. Para conocer el estado de los puertos, el atacante envía paquetes *SYN* a los puertos de la máquina objetivo y en función de la respuesta que se obtiene podemos conocer el estado de cada uno de los puertos. Las respuestas pueden ser tres, si es un paquete *SYN/ACK* como hemos visto en la negociación de tres pasos, el puerto está abierto. En cambio si la respuesta es un paquete RST significa que el puerto está cerrado. La otra posibilidad es recibir un paquete ICMP *Port Unreachable*, que significa que ese puerto está protegido por un cortafuegos.

La manera en la que podemos detectar estos ataques es observar si una misma máquina está enviando paquetes sobre muchos puertos diferentes de nuestra máquina con el objetivo de conocer el estado de cada uno de los puertos.

El patrón *TCPConnectAttackPattern* nos permite detectar que un atacante está realizando un escaneo de puertos TCP en nuestra máquina con el objetivo de encontrar vulnerabilidades y ver cuáles se encuentran abiertos y cuáles cerrados, además de conocer si alguno de ellos se encuentra protegido mediante un sistema de *firewall*. Nuestra aplicación va a detectar este ataque cuando en el periodo de un minuto se reciban 10 eventos que cumplan con las siguientes condiciones:

- La dirección ip de origen es la misma en todos los eventos.
- La dirección ip de destino es la misma en todos los eventos.
- El puerto de origen es el mismo en todos los eventos.

- El puerto de destino es distinto en cada uno de los eventos.

Con estos datos podemos asegurar que una máquina remota está enviando paquetes a los diferentes puertos de nuestro sistema. Se va a generar un informe con los datos necesarios en los que se incluya la dirección ip del atacante.

El responsable de la seguridad del sistema recibirá este informe y podrá actuar de la manera que crea conveniente para prevenir al sistema de los riesgos que conlleva esta situación.

5.5.5.7. Ataque *Flood* al email

Un ataque que también se suele realizar bastante es el de *flood* o inundación a alguno de los puertos utilizados en las conexiones por email. El objetivo del atacante en este caso es conseguir caer el puerto que se utiliza para el sistema de correo electrónico que se use en cada caso y aprovechar esa vulnerabilidad para acceder al sistema.

Para realizar estos ataques el atacante puede enviar multitud de paquetes al puerto deseado o utilizar algún software conocido como *mail bomber* que lo que hacen es enviar una gran cantidad de correos electrónicos a la dirección indicada.

Los puertos que entran en juego a la hora de realizar este ataque son 25, 143, 993 y 110. El puerto 25 es el puerto de SMTP, es decir, el protocolo simple de transferencia de correo. Este protocolo se utiliza en el intercambio de información a través del correo electrónico y está basado en un modelo cliente-servidor donde el cliente envía mensajes a uno o varios destinatarios. El puerto 143 es el puerto de IMAP, que es el protocolo de red encargado del acceso a los mensajes que se almacenan en un servidor, gracias a este protocolo se puede tener acceso al correo electrónico desde cualquier ordenador que posea conexión a internet, es decir, se permite la visualización de los correos de manera remota. Muy relacionado con este protocolo es el puerto 993, ya que es el puerto de IMAP sobre SSL, que es la capa de conexión segura, y es

un protocolo criptográfico para las comunicaciones seguras que se realizan a través de internet, en este caso, aplicadas al protocolo IMAP para el correo electrónico. El último puerto implicado en este ataque se el 110, que es el puerto de POP3, que es el protocolo encargado de obtener para los clientes locales los mensajes de correos electrónicos que se almacenan remotamente en un servidor. Este protocolo trabaja a nivel de aplicación en OSI.

Por lo tanto, para detectar estos ataques tendremos que monitorizar estos puertos y ver la actividad sospechosa que se puedan dar alrededor de ellos, con el objetivo de detectar de manera temprana a un atacante que intente realizar una inundación en los puertos de correo electrónico de nuestro sistema.

El patrón *FLOODAttackPattern* nos permite detectar ataques en los que un atacante o varios de ellos van a enviar paquetes de manera repetida sobre los puertos dedicados al correo electrónico o bien están enviando correos de manera masiva con el objetivo de bloquear estos puertos. Vamos a detectar un ataque de este tipo cuando en el periodo de un minuto se encuentren 10 eventos con las siguientes características:

- La dirección ip de destino es la misma en todos los eventos.
- El puerto de destino es el 25 o el 110 o el 143 o el 993.

Con esta información podemos comprobar que o bien un atacante o varios están realizando un ataque de inundación a nuestra máquina sobre los puertos destinados a los protocolos SMTP, IMAP, IMAP sobre SSL ó POP3.

El *listener* correspondiente va a generar un informe que será enviado al responsable de la seguridad del sistema y será el encargado de tomar las medidas necesarias para evitar consecuencias que pongan en riesgo la integridad del sistema.

5.5.5.8. Ataque al FTP

El último de los ataques comunes que vamos a identificar es el ataque al FTP mediante fuerza bruta, es decir, este ataque está basado en el envío

masivo de peticiones al puerto de FTP con el fin de colgarlo y aprovechar esa vulnerabilidad.

FTP es el protocolo de red que se utiliza para la transferencia de archivos entre diferentes sistemas que estén conectados a una red TCP. Este sistema está basado en la arquitectura cliente-servidor ya que un cliente puede acceder a un servidor donde se encuentra alojada la información y tiene la posibilidad de descargarse esta información existente o de subir nueva información. Este protocolo es independiente del sistema operativo.

Uno de las vulnerabilidades que los atacantes intentan explotar referentes al FTP es que para aprovechar la máxima velocidad este protocolo pierde en seguridad, ya que el intercambio que se hace incluido el nombre de usuario y su contraseña se realizan sin cifrado. El puerto donde se encuentran los datos es el puerto 20, pero el que debemos monitorizar con el objetivo de detectar los ataques es el puerto 21, que es el encargado de dar el control del protocolo FTP. En caso de que un atacante consiguiera colgar este puerto, podría tomar el control del protocolo y acceder a los datos.

El patrón *FTPAttackPattern* nos permite detectar ataques en los que uno o varios atacantes intentan bloquear el puerto dedicado al FTP mediante la fuerza bruta enviando masivamente paquetes. Vamos a detectar esta situación cuando en el periodo de un minuto se reciban 10 eventos que cumplan las siguientes condiciones:

- La dirección ip de destino es la misma en todos los eventos.
- El puerto de destino es el 21.

Con estos datos podemos afirmar que uno o varios atacantes están intentando inundar el puerto 21 relativo al control del protocolo de transferencia de ficheros. En este caso se va a generar un informe con los datos necesarios para que el responsable de la seguridad pueda responder de manera efectiva al ataque.

5.6. Implementación

Un buen análisis del problema y de la situación actual y un diseño claro del sistema no garantizan que a la hora de la implementación no nos vayamos a encontrar con situaciones complejas de solucionar. Por lo tanto, en esta sección vamos a mostrar cuáles son los detalles de la implementación en las diferentes partes de la aplicación, tanto de los patrones de eventos como de las clases necesarias para la integración de las diferentes tecnologías o para realizar las operaciones de filtrado, comunicación...

5.6.1. Clases de los eventos

La clase principal de nuestro sistema alrededor de la que funciona toda la aplicación son los eventos. Estos eventos son los que hemos obtenido después de realizar una transformación de los ficheros de alertas generados por *Snort* y son los que van a estar navegando por el flujo de eventos para su posterior procesamiento en el motor de CEP. Vamos a ver en el listado 5.1 como tenemos implementada esta clase para observar que atributos posee y de qué tipo son.

Listado 5.1: Clase `Event`

```
public class Event
{
   private String timestamp;
   private String ipOrigen;
   private int puertoOrigen;
   private String ipDestino;
   private int puertoDestino;

   // Metodos get y set
}
```

Esta es la clase evento con la que trabajaremos, está compuesta por cinco atributos, el primero de ellos es una cadena de caracteres llamada "*timestamp*", hace referencia al momento en horas, minutos, segundos y milisegundos en los que se ha producido el evento. Este atributo ha sido modificado de los datos que obtenemos de los ficheros de alerta de *Snort* ya que en su versión original nos incluía información referente al día y al mes en el que se producía

y eso es una información que no es necesaria para el software que estamos implementando.

Podemos observar otras dos cadenas de caracteres, en este caso son "ipOrigen" correspondiente a la dirección ip de la máquina que realiza el ataque y el atributo "ipDestino" que se corresponde con la dirección ip de la máquina que recibe dicho ataque.

También tenemos dos atributos que son números enteros, estos son "puertoOrigen" y "puertoDestino" que corresponden con el puerto que el atacante ha utilizado para atacarnos y el puerto objetivo de nuestra máquina que está siendo atacado.

Todos los atributos deben comenzar por letra minúscula y cuando el nombre está formado por más de una palabra, estas se distinguirán porque el comienzo de cada una de ellas se pone en mayúsculas, pero no se utiliza ningún signo de puntuación para diferenciarlas. Posteriormente vamos a tener los métodos *set* y *get* para cada uno de los atributos y todos seguirán el formato que se ve en el siguiente listado 5.2

Listado 5.2: Metodos `set y get`

```
public String getIpOrigen() {
      return ipOrigen;
}

public void setIpOrigen(String ipO) {
      ipOrigen = ipO;
}
```

En todos los casos tanto el método *set* como el método *get* van a comenzar por letra minúscula y el inicio de cada palabra que compone el nombre se distinguirá mediante una letra mayúscula.

Es muy importante que sigamos este formato y que tengamos implementados todos los métodos ya que si no se hace así a la hora de realizar el procesamiento complejos de eventos éste nos va a fallar cuando intentemos acceder a los atributos de los eventos que se estén procesando.

Esta clase de eventos es la clase principal con la que vamos a trabajar, en un primer momento se recibe un fichero de alerta de *Snort*, y cuando filtremos su información y realicemos la transformación se creará un objeto de java POJO que será un evento de este tipo y se incluirá en el flujo de eventos. En este flujo es donde se observará el tipo de los eventos y si el evento que llega al filtro es uno de este tipo se enviará al motor de CEP, en caso contrario se desestimará y se comprobará el tipo del siguiente evento.

A continuación vamos a ver en que situaciones vamos a utilizar esta clase y cuándo vamos a crear un evento de estas características. También veremos la manera en la que vamos a realizar el filtrado de eventos.

5.6.2. Integración, transformadores y filtrado

Ya hemos visto la clase principal de la aplicación pero ahora tenemos que ver otras clases con las que hemos desarrollado las opciones para integrar algunos componentes, hemos realizado las transformaciones y el filtrado.

Para integrar en primer lugar *Snort* con Mule ESB hemos utilizado un componente que nos proporciona el ESB del tipo *endpoint*, concretamente el componente File. A la hora de realizar la configuración XML de este componente MuleStudio nos proporciona una interfaz gráfica en la que podemos incluir los parámetros que creamos convenientes para que el funcionamiento del componente sea óptimo.

Para este tipo de componentes debemos incluir la siguiente información. Un "*Display Name*" que será el nombre que se le dará al elemento en el flujo de MuleStudio y con el que podremos referenciarlo en otras clases y en el fichero de configuración XML. También tendremos que completar toda la información en relación a la ruta que van a seguir los archivos. Por una parte debemos indicar el "*Path*" que es la ruta donde *Snort* va a generar los ficheros de alertas cada vez que detecte una situación anómala en la red. Por otra parte tendremos que incluir una ruta para la opción "*Move to directory*", con esto lo que vamos a indicar es que queremos realizar una copia de seguridad del fichero de alerta para que una vez sea consumido por el ESB enviándolo

al transformador, no perdamos la información relativa a las alertas. Gracias a esto en cualquier momento se pueden consultar estas alertas y obtener así más información relativa al ataque que se ha producido. Lo último que debemos incluir es la información referente al apartado de "*Polling Information*" en la que tendremos que indicar la frecuencia de tiempo con la que Mule ESB va a comprobar si *Snort* ha producido alguna alerta. En nuestro caso tenemos que indicar una frecuencia tan alta como para que esto se realice en tiempo real, ya que no podemos esperar para leer y procesar las alertas debido a que podemos encontrarnos en una situación crítica y esperar un retraso que se produzca en esta etapa puede condenar al fracaso el resultado final de la aplicación. El otro punto en el que debemos realizar una integración es en el último componente del flujo de Mule, es decir, en el componente java donde vamos a integrar Mule ESB con el motor de CEP. Para ello vamos a implementar una clase en java que nos permita realizar esta integración. Esta clase java la podemos ver a continuación en el listado 5.3.

Listado 5.3: Clase para `integracion`

```
package mule;

import esper.EsperUtil;
import esper.eventbean.Event;

public class SendEventToEsper {

    public void sendEvents(Event e){
        EsperUtil.getService().getEPRuntime().sendEvent(e);
    }
}
```

En esta clase tendremos que importar otras con las que vamos a trabajar, una de ellas es la clase principal o clase del tipo de evento que vimos anteriormente y la otra es la clase donde vamos a configurar Esper. Esta configuración la veremos en la próxima sección de este documento.

En esta clase lo que vamos a hacer es implementar un método que invoque al servicio cuya funcionalidad es enviar al motor de CEP el parámetro que le indiquemos, en este caso el evento que ya ha pasado el filtro en el que comprobamos su tipo. Una vez que un evento es enviado al motor de CEP vamos a proceder a la identificación de los patrones de eventos mediante el procesamiento complejo de eventos.

Otra de las partes más importantes de la implementación es la relativa a las transformaciones. En el flujo del ESB vamos a realizar dos transformaciones, primero utilizaremos la transformación de Mule para obtener una cadena a partir del fichero de alerta de *Snort* y posteriormente vamos a utilizar una clase propia que hemos implementado donde transformaremos esta cadena a un evento, además en este punto vamos a realizar la operación de discriminación de la información ya que vamos a quedarnos sólo con la información de los ficheros de alerta que necesitemos. Vamos a ir comentando algunas de las partes más importantes de la implementación que hemos realizado y viendo las diferentes secciones de las que se compone esta clase. En primer lugar vamos a ver en el listado 5.4 que otras clases necesitamos utilizar.

Listado 5.4: Clases `importadas`

```
import java.util.StringTokenizer;
import java.lang.String;

import org.mule.api.transformer.TransformerException;
import org.mule.transformer.AbstractTransformer;
import org.mule.transformer.types.DataTypeFactory;

import esper.eventbean.Event;
```

Tenemos que diferenciar tres tipos de clases externas. Por una parte clases de java como "*String*" para trabajar con cadenas o "*StringTokenizer*" [24], con esta última lo que vamos a conseguir es dividir en diferentes líneas el fichero de alertas para poder centrarnos en la línea que contiene la información necesaria para la creación del objeto POJO. Posteriormente tenemos el uso de clases propias de mule, concretamente clases necesarias para el uso de un componente de tipo *transformer*. Por último vamos a incluir la clase de los eventos de la que además usaremos sus métodos para construirlos.

En un primer lugar debemos de hacer referencia al método constructor de la clase "*StringToEvent*" que está implementado como se puede observar en el listado 5.5.

Listado 5.5: Clase `StringToEvent`

```
public StringToEvent()
{
    super();
    this.registerSourceType(DataTypeFactory.STRING);
        this.setReturnDataType(DataTypeFactory.create(Event.class)
        );
```

```
}
```

En un primer lugar vamos a indicar mediante "*super*" que este método corresponde a la superclase ya que la clase "*StringToEvent*" extiende a la clase "*AbstractTransformer*". Posteriormente indicaremos los tipos de datos con los que vamos a trabajar, *string* serán los datos fuente y la clase de eventos que definimos anteriormente será el tipo de datos correspondiente al resultado de esta clase.

La transformación la vamos a realizar en la clase "*doTransform*" que tiene la cabecera que vamos a ver en el listado 5.6.

Listado 5.6: Clase `doTransform`

```
public Object doTransform(Object src, String encoding) throws
TransformerException
```

En esta línea estamos indicando que vamos a crear un objeto que finalmente será un POJO y que vamos a tratar un un objeto con una codificación que se le ha indicado en el transformador. A partir de aquí tenemos un objeto "*src*" con el que vamos a poder leer el fichero de alerta.

El siguiente paso será leer el fichero de alerta, utilizar la clase *Tokenizer* para separarlos en líneas y quedarnos únicamente con la línea que contiene la información relativa a los datos con los que vamos a crear el evento. Se puede ver el código en el listado 5.7

Listado 5.7: Uso de `tokenizer`

```
String f = (String) src;

int i=0;

StringTokenizer tokens = new StringTokenizer(f,"\n");
int nDatos=tokens.countTokens();
    String[] datos=new String[nDatos];

while(tokens.hasMoreTokens()){
    String linea = tokens.nextToken();
    datos[i]=linea;
    tokens.nextToken();
    i++;
    }

String linea = datos[1];
```

En la primera línea vamos a realizar una transformación del objeto "*src*" a cadena de caracteres y posteriormente podemos ver el uso de los *tokens* utilizando la clase "*Tokenizer*", donde vamos a separar las líneas utilizando como elemento discriminatorio el salto de línea y una vez que sepamos el número de ellas que tenemos vamos a crear un *array* para cada una de ellas. En última instancia nos vamos a quedar únicamente con la línea que necesitamos usar ya que posee toda la información relativa a los ataques.

Ahora tendremos que establecer las diferentes separaciones entre los atributos para discriminar la información que no es necesaria de esta línea y poder construir las subcadenas con los datos que vamos a almacenar en los atributos de la clase. Vamos a ver un ejemplo en el listado 5.8 de como establecemos las diferentes separaciones para obtener el *timestamp* y como creamos ese atributo.

Listado 5.8: Seleccion del `timestamp`

```
int sep1 = linea.indexOf("-");
int sep2 = linea.indexOf("␣");

String timestamp = linea.substring(sep1+1,sep2);
```

Utilizamos la función "*indexOf*" que nos devuelve la posición del elemento que le pasemos por parámetro y posteriormente gracias a la función "*substring*" podemos crear un atributo de tipo cadena de caracteres. En el caso de los puertos de origen y destino, como su tipo no son cadenas sino enteros tendremos que realizar una operación más que vamos a ver en el listado 5.9.

Listado 5.9: Transformacion para `puertoOrigen`

```
int puertoOrigen = Integer.parseInt(puertoOrigenS);
```

Podemos observar como utilizamos una operación de transformación para obtener un entero a partir de una cadena de caracteres.

La última parte de la clase es la creación del evento y de sus atributos para posteriormente devolverla. Podemos verla en el listado 5.10

Listado 5.10: Creacion de un `evento`

```
Event e = new Event();

e.setTimestamp(timestamp);
e.setIpOrigen(ipOrigen);
```

```
e.setPuertoOrigen(puertoOrigen);
e.setIpDestino(ipDestino);
e.setPuertoDestino(puertoDestino);

return e;
```

En este punto ya tenemos un evento que se corresponde con una alerta que ha generado *Snort*, ya hemos visto el proceso de transformación desde un fichero de *Snort* hasta un objeto POJO que será una alerta que se introducirá en el flujo de eventos para su posterior filtrado y procesamiento en el motor de CEP.

Ahora vamos a ver como podemos realizar el filtrado de eventos en el flujo gracias a un componente de Mule ESB, concretamente al uso de un "*payload*" que es un filtro en el que le vamos a indicar que sólo permita el paso de aquellos eventos que son iguales a los eventos que hemos definido en la clase principal.

Esto es necesario ya que en un sistema podemos tener multitud de eventos y estamos ante una aplicación escalable y que puede ser fácilmente ampliada, en este caso se le podría introducir otro tipo de eventos que completarán a los que genera *Snort* para la detección de otros ataques distintos, en este caso sería obligado el uso de un filtro para conocer que tipo de eventos podemos dejar pasar al motor de CEP. La configuración en el fichero de XML es la que podemos ver en el listado 5.11

Listado 5.11: Clase `payload`

```
<payload-type-filter expectedType="esper.eventbean.Event" doc:
    name=Event-Filter-Type"/>
```

Podemos observar como le indicamos que el tipo de los eventos a los que debe permitir el paso son aquellos similares a los de la clase "*esper.eventbean.Event*", desestimando los demás y no enviándolos al motor de CEP.

Una vez visto como podemos realizar las transformaciones, los filtrados y las integraciones de las diferentes tecnologías vamos a ver como podemos configurar el motor de CEP para realizar el procesamiento de eventos complejos.

5.6.3. Configuración del motor de CEP

En este punto debemos explicar cómo hemos realizado configuración de Esper que es el motor de CEP que vamos a utilizar. La secuencia que vamos a seguir para la detección de ataques de seguridad es la siguiente:

1. Configuración e inicialización del motor Esper.
2. Generación de alertas a través de *Snort*.
3. Transformación a eventos e introducción de éstos en el flujo de eventos.
4. Registro de los patrones de eventos complejos en el motor.
5. Análisis de los eventos que llegan hasta el motor e identificación de eventos complejos.
6. Envío de la información al *listener* correspondiente.
7. Comunicación con el responsable de la seguridad a partir del informe generado en el *listener*.

Vamos a ver la implementación que hemos realizado para la configuración e inicialización del motor Esper, para ello en primer lugar vamos a ver en el listado 5.12 que clases propias de este motor debemos instanciar.

Listado 5.12: Clases `instanciadas`

```
import com.espertech.esper.client.Configuration;
import com.espertech.esper.client.EPServiceProvider;
import com.espertech.esper.client.EPServiceProviderManager;
import com.espertech.esper.client.EPStatement;
```

Con la primera de ellas vamos a obtener una instancia que representa todos los parámetros de configuración del motor, con las demás vamos a tener la posibilidad de trabajar con las interfaces administrativas y de ejecución para la instancia del motor.

Las líneas más importantes de la configuración del motor son las que se ven en el listado 5.13.

Listado 5.13: Configuracion del `motor`

```
Configuration config = new Configuration();
config.addImport("esper.eventbean.Event");
config.addEventTypeAutoName("esper.eventbean.Event");
config.addEventType("Event", Event.class.getName());
epService = EPServiceProviderManager.getProvider("myCEPEngine",
    config);
```

Con esto vamos a obtener una instancia "*config*" que la hemos instanciado de manera directa y a partir de ahí le vamos a añadir los valores necesarios. Posteriormente esta instancia va a pasar a "*EPServiceProviderManager*" dónde vamos a obtener un motor Esper configurado correctamente.

A partir de este momento debemos definir cuáles van a ser los patrones de eventos que vamos a detectar e identificar sus *listener* correspondientes. Un ejemplo de esto se puede ver en el listado 5.14.

Listado 5.14: Patron `DOS`

```
DOSAttackPattern dos = new DOSAttackPattern(epService.
    getEPAdministrator());
dos.addListener(new DOSAttackPatternListener());
```

En este caso estamos estableciendo lo necesario para detectar un ataque de denegación de servicio y su correspondiente *listener*. En la siguiente sección vamos a ver la implementación de todos estos patrones de eventos y posteriormente veremos como implementar los *listener* adecuados para ellos.

5.6.4. Patrones de eventos complejos

Vamos a mostrar la implementación que hemos realizado para cada uno de los ataques que vamos a detectar.

5.6.4.1. Ataque DOS

La implementación en EPL para detectar un ataque de denegación de servicio es la que se puede ver en el listado 5.15

Listado 5.15: Patron `DOS`

```
String DOSAttackPattern = "@Name('DOSAttackPattern')
```

```
insert into DOSAttack
select SuspectedDOSAttack.timestamp as timestamp,
   SuspectedDOSAttack.ipOrigen as ipO,
   SuspectedDOSAttack.ipDestino as ipD,
        SuspectedDOSAttack.puertoOrigen as po,
        SuspectedDOSAttack.puertoDestino as pd
from pattern [every SuspectedDOSAttack = Event
   -> [9] Event(ipOrigen=SuspectedDOSAttack.ipOrigen,
   ipDestino = SuspectedDOSAttack.ipDestino,
   puertoDestino = SuspectedDOSAttack.puertoDestino)
where timer:within(1 min)]";
```

Podemos ver como se define el patrón con la cláusula *from pattern* y tomamos de todos los eventos del flujo aquellos que cumplan con las condiciones que vimos en la sección 5.5.5.1 y aplicamos el operador *every* para seleccionar cada uno de estos eventos que sean sospechosos de ser un ataque DOS, en nuestro caso lo hemos denominado *SuspectedDOSAttack*.

Podemos ver en la cláusula *select* como cogemos todos los atributos que necesitamos y le asignamos un alias mediante la palabra *as* y utilizando la palabra reservada *insert into* lo metemos en el flujo que hemos denominado *DOSAttack*.

Para crear el informe que mostraremos por pantalla y enviaremos al responsable de la seguridad enviamos esta información al *listener* correspondiente del cuál veremos su implementación en la siguiente sección.

Para que quede más claro podemos ver esta implementación en la figura 5.13

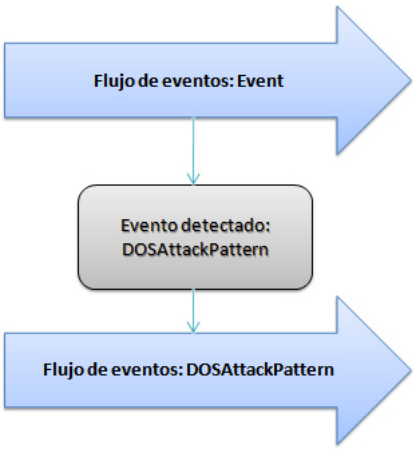


Figura 5.13: Patrón Ataque DOS

5.6.4.2. Ataque DDOS

La implementación del patrón relativo al ataque distribuido de denegación de servicio en EPL la podemos ver en el listado 5.16

Listado 5.16: Patron `DDOS`

```
String DDOSAttackPattern = "@Name('DDOSAttackPattern')
insert into DDOSAttack
select SuspectedDDOSAttack.timestamp as timestamp,
   SuspectedDDOSAttack.ipOrigen as ipO,
   SuspectedDDOSAttack.ipDestino as ipD,
   SuspectedDDOSAttack.puertoOrigen as po,
   SuspectedDDOSAttack.puertoDestino as pd
from pattern[every SuspectedDDOSAttack = Event
   -> [9] Event(ipOrigen != SuspectedDDOSAttack.ipOrigen,
   ipDestino = SuspectedDDOSAttack.ipDestino,
   puertoDestino = SuspectedDDOSAttack.puertoDestino)
where timer:within(1 min)]";
```

Podemos ver en la implementación el patrón definido en la cláusula *from pattern* y como de todos los eventos que se envían a través del flujo tomamos aquellos que cumplan las condiciones que vimos en la sección 5.5.5.2, posteriormente aplicamos el operador *every* con el cual seleccionamos cada uno de estos eventos que puedan ser sospechosos de ser un ataque DDOS, en nuestra aplicación los hemos llamado *SuspectedDDOSAttack*.

En la cláusula *select* vamos a seleccionar los atributos necesarios para su posterior procesamiento y les vamos a asignar un alias mediante el operador *as*. Finalmente con la cláusula *insert into* lo añadiremos al flujo que hemos llamado *DDOSAttack*.

Posteriormente enviaremos esta información al *listener* correspondiente donde generaremos un informe que mostraremos por pantalla y enviaremos al responsable de la seguridad mediante un email.

En la figura 5.14 podemos ver de manera gráfica la implementación del patrón.

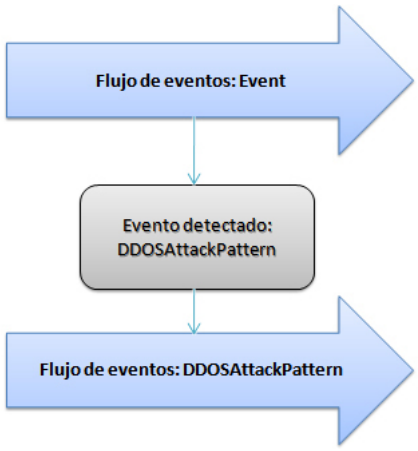


Figura 5.14: Patrón Ataque DDOS

5.6.4.3. Ataque Smurf

Vamos a ver en el listado 5.17 la implementación en EPL para el patrón relativo al ataque smurf:

Listado 5.17: Patron `SMURF`

```
String SmurfAttackPattern = "@Name('SmurfAttackPattern')
insert into SmurfAttack
select SuspectedSmurfAttack.timestamp as timestamp,
   SuspectedSmurfAttack.ipOrigen as ipO,
   SuspectedSmurfAttack.ipDestino as ipD,
   SuspectedSmurfAttack.puertoOrigen as po,
   SuspectedSmurfAttack.puertoDestino as pd
from pattern[every SuspectedSmurfAttack = Event
```

```
    -> [1] Event(ipOrigen = ipDestino,
    ipDestino = SuspectedSmurfAttack.ipDestino,
    puertoOrigen != SuspectedSmurfAttack.puertoDestino)
where timer:within(1 min)]";
```

Como podemos comprobar, hemos definido el patrón en la cláusula *from pattern* y hemos indicado que de todos los eventos que se transmiten a través del flujo, seleccionemos aquellos que cumplan las condiciones que hemos visto en la sección 5.5.5.3, después vamos a aplicar el operador *every* para escoger todos estos ataques que puedan ser sospechosos de ser un ataque smurf y los hemos denominado *SuspectedSmurfAttack*.

En la cláusula *select* vamos a elegir aquellos atributos necesarios asignándoles un alias mediante la palabra reservada as y con la cláusula *insert into* lo vamos a incluir en el flujo al que hemos denominado *SmurfAttack*.

Después vamos a enviar esta información a su correspondiente *listener* en donde vamos a generar un informe que enviaremos al responsable de la seguridad mediante un email y además mostraremos por pantalla.

Podemos ver de manera más clara la implementación en la figura 5.15

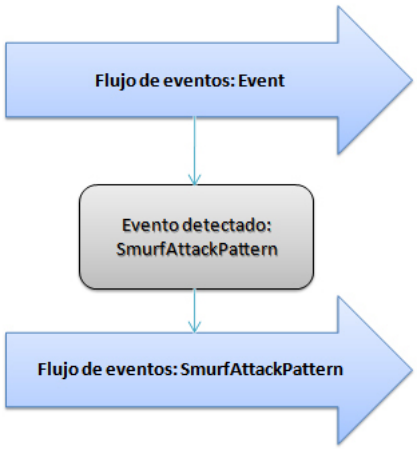


Figura 5.15: Patrón Ataque Smurf

5.6.4.4. Ataque Land

Vamos a ver en el listado 5.18 la implementación en EPL del ataque *land*

Listado 5.18: Patron `LAND`

```
String LandAttackPattern = "@Name('LandAttackPattern')
insert into LandAttack
select SuspectedLandAttack.timestamp as timestamp,
   SuspectedLandAttack.ipOrigen as ipO,
   SuspectedLandAttack.ipDestino as ipD,
   SuspectedLandAttack.puertoOrigen as po,
   SuspectedLandAttack.puertoDestino as pd
from pattern[every SuspectedLandAttack = Event
   -> [1] Event(ipOrigen=SuspectedLandAttack.ipDestino
   ipDestino = SuspectedLandAttack.ipDestino,
   puertoOrigen = SuspectedLandAttack.puertoDestino,
   puertoDestino = 113 OR puertoDestino = 139)
where timer:within(1 min)]";
```

En el código que acabamos de ver, podemos identificar el patrón en la cláusula *from pattern* y como indicamos que seleccionemos de todos los eventos que se transmiten a través del flujo, sólo aquellos que cumplan las condiciones que establecimos en la sección 5.5.5.4 Posteriormente vamos a aplicar el operador *every* que nos servirá para elegir los ataque que puedan ser sospechosos de ser un ataque *land* y le ponemos el nombre *SuspectedLandAttack.*

En la cláusula *sel*ect vamos a seleccionar los atributos necesarios para su posterior procesamiento y utilizaremos la palabra reservada *as* para asignarles un alias, posteriormente lo insertaremos en el flujo *LandAttack* utilizando la cláusula *insert into.*

Luego vamos a enviar esa información al *listener* correspondiente para generar un informe que imprimiremos por pantalla además de ser enviado al responsable de la seguridad del sistema.

Vamos a ver esta implementación de manera más clara en la figura 5.16

Figura 5.16: Patrón Ataque Land

5.6.4.5. Ataque Supernuke

Vamos a ver en el listado 5.19 de que manera hemos implementado mediante EPL el patrón referente al ataque *supernuke*

Listado 5.19: Patron `SUPERNUKE`

```
String SupernukeAttackPattern = "@Name('SupernukeAttackPattern')
insert into SupernukeAttack
select SuspectedSupernukeAttack.timestamp as timestamp,
   SuspectedSupernukeAttack.ipOrigen as ipO,
   SuspectedSupernukeAttack.ipDestino as ipD,
   SuspectedSupernukeAttack.puertoOrigen as po,
   SuspectedSupernukeAttack.puertoDestino as pd
from pattern[every SuspectedSupernukeAttack = Event
   -> [9] Event(ipDestino = SuspectedSupernukeAttack.ipDestino,
   puertoDestino = 137 OR puertoDestino = 138 OR puertoDestino =
       139)
where timer:within(1 min)]";
```

Como podemos ver en la implementación, vamos a utilizar las palabras reservadas *from pattern* para identificar al patrón y vamos a seleccionar del flujo de eventos aquellos que cumplan las condiciones que hemos incluido en la sección 5.5.5.5 para después utilizar el operador *every* gracias al cuál vamos a poder escoger cuáles son los ataques sospechosos de ser un ataque *supernuke* y los vamos a denominar *SuspectedSupernukeAttack*.

Vamos a utilizar la cláusula *select* para escoger que atributos vamos a necesitar posteriormente y les pondremos un alias utilizando el operador *as*. Después vamos a insertar mediante la cláusula *insert into* este ataque en el flujo *SupernukeAttack*.

Para finalizar enviaremos toda la información que hemos seleccionado al correspondiente *listener* y ahí se creará un informe que se imprimirá por pantalla y se enviará al responsable de la seguridad del sistema.

Podemos ver de manera más clara esta implementación en la figura 5.17

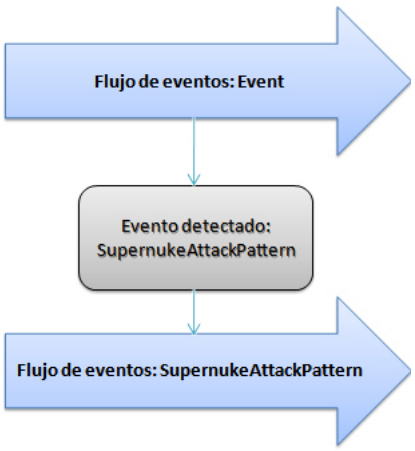


Figura 5.17: Patrón Ataque Supernuke

5.6.4.6. Ataque Escaneo de puertos TCP

Vamos a ver en el listado 5.20 la implementación en EPL relativa al patrón que define un escaneo de puertos TCP

Listado 5.20: Patron `Escaneo Puertos TCP`

```
String TCPConnectAttackPattern = "@Name('TCPConnectAttackPattern
    ')
insert into TCPConnectAttack
select SuspectedTCPConnectAttack.timestamp as timestamp,
   SuspectedTCPConnectAttack.ipOrigen as ipO,
   SuspectedTCPConnectAttack.ipDestino as ipD,
   SuspectedTCPConnectAttack.puertoOrigen as po,
   SuspectedTCPConnectAttack.puertoDestino as pd
```

```
from pattern[every SuspectedTCPConnectAttack = Event
    -> [9] Event(ipOrigen=SuspectedTCPConnectAttack.ipOrigen,
    ipDestino = SuspectedTCPConnectAttack.ipDestino,
    puertoOrigen = SuspectedTCPConnectAttack.puertoOrigen,
    puertoDestino != SuspectedTCPConnectAttack.puertoDestino)
where timer:within(1 min)]";
```

Si observamos la implementación de este patrón podemos observar que lo identificamos mediante las palabras reservadas *from pattern*. Posteriormente seleccionaremos aquellos eventos del flujo de eventos que cumplan las condiciones que hemos establecido con anterioridad en la sección 5.5.5.6 y después vamos a utilizar el operador *every* para indicar que eventos son sospechosos de ser un escaneo de puertos y los denominaremos *SuspectedTCPConnectAttack*.

Utilizaremos la cláusula *select* para seleccionar aquellos atributos que vamos a necesitar después y mediante la palabra reservada as le pondremos un alias. Posteriormente los insertaremos en el flujo *TCPConnectAttack* gracias a la cláusula *insert into*.

Finalizaremos con el envío de esta información al *listener* correspondiente dónde se va a generar el informe que se va a mostrar por pantalla y se va a enviar vía email al responsable de la seguridad del sistema.

Podemos ver en la figura 5.18 esta implementación.

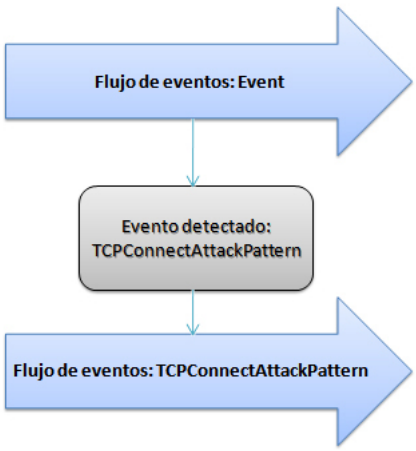


Figura 5.18: Patrón Ataque Escaneo de puertos TCP

5.6.4.7. Ataque Flood al email

Vamos a ver en el listado 5.21 la implementación en EPL del patrón relativo al ataque mediante flood al email

Listado 5.21: Patron `FLOOD`

```
String FLOODAttackPattern = "@Name('FLOODAttackPattern')
insert into FLOODAttack
select SuspectedFLOODAttack.timestamp as timestamp,
   SuspectedFLOODAttack.ipOrigen as ipO,
   SuspectedFLOODAttack.ipDestino as ipD,
   SuspectedFLOODAttack.puertoOrigen as po,
   SuspectedFLOODAttack.puertoDestino as pd
from pattern[every SuspectedFLOODAttack = Event
   -> [9] Event(ipDestino = SuspectedFLOODAttack.ipDestino,
   puertoDestino = 25 OR puertoDestino = 110
   OR puertoDestino = 143 OR puertoDestino = 993)
where timer:within(1 min)]";
```

Como podemos ver en la implementación vamos a utilizar la cláusula *from pattern* para identificar el patrón, y vamos a escoger aquellos eventos del flujo de eventos que cumplan las condiciones que hemos indicado en la sección 5.5.5.7 de este documento, posteriormente con el uso del operador *every* vamos a especificar cuáles de estos eventos son sospechosos de ser un ataque mediante flood al email y los nombraremos *SuspectedFLOODAttack*.

En la cláusula *select* vamos a indicar que atributos vamos a necesitar para trabajar posteriormente con ellos, utilizaremos la cláusula *as* para ponerles un alias y los insertaremos en el flujo *FLOODAttack* mediante el uso de la cláusula *insert into*.

Posteriormente vamos a enviar la información al *listener* que será el encargado de generar el informe que se imprimirá en pantalla y se enviará por email al responsable de la seguridad.

En la figura 5.19 podemos ver de manera más clara la implementación de este patrón.

Figura 5.19: Patrón Ataque Flood al email

5.6.4.8. Ataque al FTP

A continuación en el listado 5.22 vamos a ver la implementación del patrón que identifica a un ataque sobre el FTP de nuestra máquina

Listado 5.22: Patron `FTP`

```
String FTPAttackPattern = "@Name('FTPAttackPattern')
insert into FTPAttack
select SuspectedFTPAttack.timestamp as timestamp,
   SuspectedFTPAttack.ipOrigen as ipO,
   SuspectedFTPAttack.ipDestino as ipD,
   SuspectedFTPAttack.puertoOrigen as po,
   SuspectedFTPAttack.puertoDestino as pd
from pattern[every SuspectedFTPAttack = Event
   -> [9] Event(ipDestino = SuspectedFTPAttack.ipDestino,
       puertoDestino = 21)
where timer:within(1 min)]";
```

Podemos observar en la implementación del patrón que lo hemos definido en la cláusula *from pattern*, posteriormente vamos a elegir los eventos del flujo de eventos que cumplan las condiciones que establecimos en la sección 5.5.5.8 y después vamos a utilizar el operador *every* para indicar que eventos de este flujo son sospechosos de ser un ataque al FTP y los vamos a llamar *SuspectedFTPAttack*.

Posteriormente vamos a utilizar la palabra reservada *select* para especificar los atributos con los que trabajaremos luego y les pondremos un alias utilizando el operador *as*. Mediante la cláusula *insert into* vamos a insertar estos eventos en el flujo *FTPAttack*.

Para finalizar enviaremos la información relativa al ataque al correspondiente *listener* para que genere el informe que se mostrará en la pantalla y se enviará vía email al responsable de la seguridad.

Vamos a ver mejor la implementación del patrón en la figura 5.20

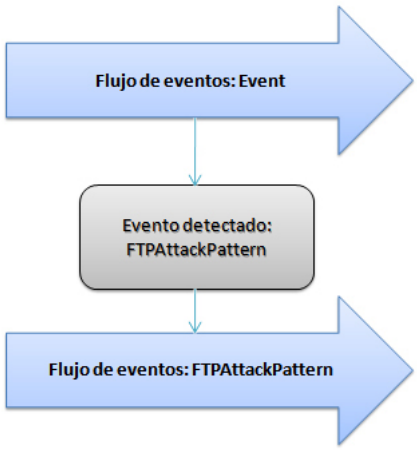


Figura 5.20: Patrón Ataque al FTP

5.6.5. Listeners

En esta sección vamos a ver como se realiza la implementación de un *listener*. Cada patrón de evento que hemos diseñado lleva asociado un *listener* que recibe la información cada vez que se identifica un ataque.

En primer lugar tenemos que ver que clases son las que van a ser utilizadas en la implementación. Vamos a verlo en el listado 5.23

Listado 5.23: Clases importadas en `Listener`

```
import com.espertech.esper.client.EventBean;
import com.espertech.esper.client.UpdateListener;
import mule.SendMail;
```

Podemos observar que necesitamos utilizar la clase referente a los eventos y la clase *UpdateListener* que nos sirve para realizar la actualización en el *listener* con los datos que van llegando a partir del ataque que se ha identificado con el patrón en EPL. También debemos usar la clase "*SendMail*" que hemos implementado nosotros y será la encargada de enviar por email la información al responsable de la seguridad. En la próxima sección veremos su implementación.

La parte más importante de los *listener* es la relativa a la creación del informe del ataque que se ha producido y su envío al responsable de la seguridad, vamos a ver en el listado 5.24 de que manera lo hemos realizado.

Listado 5.24: Clase de `Listener`

```
String alerta = "Ataque DOS detectado a las: " + (String)
    newEvents[0].get("timestamp")
"en la IP: " + (String) newEvents[0].get("ipD")
"y en el puerto: " + (Integer) newEvents[0].get("pd")
"Proveniente de la IP: " + (String) newEvents[0].get("ipO")
"y del puerto: " + (Integer) newEvents[0].get("pd");

SendMail.EnvioAlerta(alerta);
```

Podemos observar como generamos un informe relativo al ataque producido utilizando los atributos que se han seleccionado previamente en el patrón en EPL. Posteriormente utilizamos la clase *SendMail* con su operación *EvioAlerta* para avisar al responsable de la seguridad en tiempo real del ataque que se ha producido.

5.6.6. Comunicación con el usuario final

Esta es la última parte del flujo que se produce desde el momento en el que *Snort* genera una alerta hasta cuando nuestra aplicación cierra el ciclo con el envío de la información relativa al ataque al responsable de la seguridad del sistema.

Para este fin hemos diseñado una clase que hemos denominado *SendMail* y que utiliza las operaciones que nos proporciona el paquete *javax.mail* que se puede usar de manera libre.

En un primer momento tenemos que establecer las propiedades para el envío del email, para ello debemos escribir la línea que vemos en el listado 5.25

Listado 5.25: Propiedades `clase SendMail`

```
Properties props = new Properties();
```

A partir de aquí debemos establecer el *host*, el nombre de usuario y la autenticación. Después debemos utilizar estas propiedades para establecer la sesión como se ve en el listado 5.26

Listado 5.26: Establecer `sesion`

```
Session session = Session.getDefaultInstance(props);
```

Una vez establecida la sesión, debemos especificar los datos del responsable de seguridad al que vamos a enviar el informe. Después debemos crear dicho mensaje con el informe que hemos pasado desde el *listener* correspondiente de la manera que se ve en el listado 5.27

Listado 5.27: Crear `mensaje`

```
message.setText(alerta);
```

Por último tenemos que enviar el mensaje, utilizaremos la función *getTransport* y lo enviaremos como se ve en el listado 5.28

Listado 5.28: Enviar `Mensaje`

```
Transport t = session.getTransport("smtp");
t.sendMessage(message, message.getAllRecipients());
```

De esta manera una vez que se ha producido el ataque el sistema ha enviado un informe detallando la alarma y teniendo el responsable de la seguridad la posibilidad de tomar la decisión que crea conveniente.

5.7. Pruebas

En una aplicación como esta que trata un campo de vital importancia es indispensable probar todos los casos y ataques posibles para comprobar que el sistema es capaz detectarlos y enviar las alarmas generadas al responsable en tiempo real.

Para realizar las pruebas no hemos utilizado ninguna plataforma específicamente diseñada para tal fin, sino que hemos realizado directamente los ataques sobre el sistema con el objetivo de comprobar su funcionamiento en un entorno real.

Para ello se ha utilizado un ordenador que ha sido el objetivo de los ataques y se ha colocado en una red con otros ordenadores que han sido los que han realizado los ataques. En el ordenador uno se ha instalado y configurado *Snort* y se ha ejecutado esta aplicación. Desde los demás ordenadores se han realizado los diferentes ataques que hemos visto en la sección 5.5.5.

Para realizar los ataques hemos utilizado algunas herramientas específicas como *nmap* [17] con la que podemos realizar diferentes acciones, entre ellas realizar escaneos de puertos.

Hemos utilizado diferentes equipos para realizar los ataques que tienen como características que son distribuidos ya que son o bien diferentes atacantes o bien robots infectados por un sólo atacantes.

Para probar en cualquier momento el funcionamiento del sistema sin necesidad de realizar los ataques hemos diseñado un *script* con el cuál generamos directamente las alertas de *Snort* relativas al ataque que escojamos realizar desde el menú. Estas alertas se crean en la carpeta que utiliza *Snort* para ello simulando así el mismo funcionamiento que si los ataques se realizasen en una situación real.

*

Capítulo 6

Resumen

En este proyecto fin de carrera hemos desarrollado una herramienta dentro del "Grupo UCASE de Ingeniería del Software" (TIC-025) de la Universidad de Cádiz destinada a aumentar la seguridad de los sistemas informáticos. Su principal objetivo es detectar ataques a través de la red a un equipo informático utilizando el procesamiento de eventos complejos.

Esta herramienta se puede enmarcar dentro de los IDS basados en red y su fin es aumentar la seguridad de los equipos detectando las situaciones en tiempo real para que se pueda prevenir del ataque y evitar así accesos a la información que se encuentra almacenada en el equipo.

Se va a utilizar un *sniffer* llamado *Snort* que monitorizará el tráfico de la red para detectar así cualquier situación que pueda poner en riesgo la integridad de la seguridad de nuestro equipo. *Snort* va a generar ficheros en tiempo real relativos a la alerta detectada y nosotros los procesaremos y convertiremos en eventos para incluirlos en un flujo de eventos que llegará al motor de CEP Esper donde se detectarán ataques identificando los patrones de eventos que hemos diseñado.

Para crear esta estructura hemos seguido una arquitectura SOA 2.0 incluyendo como pieza fundamental un ESB que nos permitirá integrar las diferentes tecnologías y hacer uso de las herramientas necesarias para crear un sistema capaz de responder en tiempo real a estos ataques que se detectan.

Para ello se va realizar un informe relativo a las alarmas que genera CEP tras la identificación de un ataque y será enviado mediante un correo electrónico en tiempo real al responsable de la seguridad del sistema que será el encargado de tomar la decisión más adecuada para prevenir al sistema del ataque que ha sufrido.

Capítulo 7

Conclusiones y trabajo futuro

En este capítulo vamos a tratar las conclusiones a las que hemos llegado tras el desarrollo de esta aplicación y qué líneas podemos seguir desarrollando en relación a este proyecto.

7.1. Valoración

El proyecto que hemos desarrollado tiene como utilidad aumentar la seguridad de los sistemas informáticos tanto en las empresas como para usuarios particulares. La motivación principal para realizar este proyecto ha sido por una parte el interés en un campo tan importante como la seguridad informática y por otra, el uso de nuevas tecnologías que permiten procesar y correlacionar información en tiempo real,cada vez más demandadas como es CEP, y su integración en arquitecturas SOA 2.0. Así pues, CEP permite enviar alarmas en tiempo real cuando se producen situaciones relevantes, a partir de la detección de ciertos patrones de eventos definidos previamente, y SOA facilita la integración de sistemas de información heterogéneos.

Una vez finalizado este proyecto podemos concluir con que se han cumplido todos los objetivos propuestos y nuestro sistema es capaz de responder ante muchos de los ataques que más se producen en la actualidad.

En esta memoria se recoge una buena parte del estado actual de los sistemas de seguridad y su utilidad, además de servir como punto de partida para el desarrollo de cualquier aplicación que se desee realizar aprovechando todos los eventos que nos rodean y dotando a los sistemas de la capacidad de responder en tiempo real.

Los conocimientos más importantes que hemos adquirido a lo largo del desarrollo del proyecto han sido los siguientes:

- Arquitecturas SOA, EDA y SOA 2.0.
- Integración de diferentes tecnologías y aplicaciones mediante ESB.
- Procesamiento de eventos complejos.
- Estructura y funcionamiento de sistemas de seguridad.
- Lenguajes de programación EPL y Java.
- Herramientas como *Snort*, Eclipse, Mule y MuleStudio.

Además este proyecto ha servido como introducción a la investigación ya que se ha desarrollado una aplicación utilizando tecnologías novedosas de las que no existe mucha información y ha sido necesario un periodo importante de investigación durante todo el desarrollo del proyecto.

Podemos concluir con que se han conseguido llevar a cabo todas las expectativas puestas en este proyecto tanto a nivel técnico como a nivel personal.

7.2. Trabajo futuro

El proyecto que hemos llevado a cabo se enmarca dentro de algunas de las líneas de investigación del "Grupo UCASE de Ingeniería del Software" (TIC-025) de la Universidad de Cádiz, en concreto, las arquitecturas orientadas a servicios y las arquitecturas dirigidas por eventos. Así pues, nos encontramos ante un proyecto que podrá ser mejorado y ampliado en un futuro próximo.

Una de las mejoras que podrá introducirse es el diseño de nuevos patrones de eventos para que elsistema sea capaz de reconocer más ataques de seguridad. Otra posible mejora es el la incorporación de nuevos sensores a nuestra arquitectura que produzcan eventos de interés para nuestro escenario de seguridad. Actualmente usamos un *sniffer* como es *Snort* para el monitorizado de la red pero podemos pensar en la inclusión de nuevas fuentes de información que no tienen por qué estar directamente relacionadas con el estado de la red, sino con otras funciones del sistema, como, por ejemplo, que se detecte algún fallo en el sistema lo que nos haga pensar que se está produciendo otro tipo de ataques.

Otra línea en la que se puede trabajar en un futuro es en conseguir generar respuestas automáticas a los ataques realizando un estudio de lo que es más conveniente en cada caso particular. Y una última posibilidad de ampliación puede ser la creación de una interfaz gráfica para interaccionar de manera más amigable con la aplicación, sobre todo aquellos usuarios no familiarizados con estas tecnologías.

Como se ha comentado a lo largo de esta memoria, se ha escogido la seguridad informática como escenario en este trabajo. No obstante, este proyecto servirá de base para desarrollar otros sistemas que sean capaces de trabajar con un ESB y con información en tiempo real mediante el procesamiento de eventos complejos, utilizando para ello el motor de CEP Esper y el lenguaje EPL para el diseño de nuevos patrones de eventos.

*

Capítulo 8

Agradecimientos

- Al director del proyecto, Juan Boubeta Puig, por darme la oportunidad de realizar este proyecto y por su apoyo y consejos a lo largo de todo el desarrollo.
- A mis compañeros que me han ayudado a lo largo de toda la carrera y a los profesores de los cuáles he adquirido los conocimientos necesarios para poder desarrollar proyectos de este tipo.
- A mi familia por su apoyo en todo este tiempo.

*

Capítulo 9

Manual de instalación

Vamos a ver en este capítulo los pasos que debemos seguir para instalar nuestra aplicación en una máquina. A la hora de realizar esta instalación debemos de incluir en ella a *Snort* y MuleStudio.

9.1. Instalación de Snort

El primer paso es instalarnos *Snort* y un paquete de reglas para poder utilizarlo como sensor que monitorice el tráfico de nuestra red y detecte así las situaciones peligrosas para su posterior procesamiento.

En primer lugar es necesario descargar *Snort* de la página oficial: http://www.snort.org/start/download desde aquí se podrá obtener tanto una versión para sistemas UNIX como para Windows. También lo podemos descargar mediante comandos de la siguiente manera:

```
$ wget http://www.snort.org/dl/snort-current/<filename> -O
<output-filename>
```

Posteriormente para poder descargar el paquete de reglas que vamos a utilizar para la detección de amenazas vamos a descargarnos la aplicación *oinkmaster*. La dirección para la descarga es la siguiente: http://oinkmaster.sourceforge.net/download.shtml

Debemos registrarnos en la web de *Snort* [21] para obtener el código *oink* que utilizaremos para la descarga de los paquetes de reglas.

Para el uso de *Snort* debemos tener instalado la base de datos MySQL, PHP y Apache, para ello vamos a escribir la siguiente instrucción:

```
sudo apt-get install apache2 php5 mysql-server phpmyadmin
```

Además necesitamos diferentes bibliotecas de las que se harán uso. Para ello vamos a ejecutar la siguiente orden en la consola:

```
sudo apt-get install php5-gd php5-ldap php5-dev php5-mysql php-pear
libnet1 libnet1-dev libpcap0.8 libpcap-dev libpcap0.8-dev libpcre3
expect gobjc libpcre3-dev flex bison libmysql++-dev libapache2-mod-php5
php5-cgi -y
```

Una vez en este punto ya podemos pasar a la instalación de *Snort*. Para ello debemos descomprimir los ficheros de *Snort* que nos descargamos anteriormente y nos situaremos dentro de la carpeta descomprimida.

En este punto debemos configurar la instalación y posteriormente realizarla, para ello vamos a ejecutar los siguientes comandos:

```
# ./configure
```

```
# make
```

```
# make install
```

Una vez instalado debemos crear los directorios que se van a utilizar para almacenar las alertas y las reglas. Para ello utilizamos los siguientes comandos:

```
# sudo mkdir -p /etc/snort
```

```
# sudo mkdir -p /var/log/snort
```

```
# sudo mkdir -p /etc/snort/rules
```

Y debemos copiar los ficheros necesarios de los que disponemos en estas carpetas de la siguiente manera:

```
# cp -r preproc_rules/ /etc/snort/
```

```
# cp etc/*.conf* /etc/snort/
```

```
# cp etc/*.map /etc/snort/
```

Ahora es el momento de trabajar con las reglas. Para ello debemos descomprimir el fichero que nos descargamos de *oinkmaster* y situarnos en él. Ahora debemos realizar el registro de los códigos *oink* de *Snort*. Para ello una vez

registrados en la web como indicamos al principio, en la sección *Oinkcodes* podremos ver el código que nos corresponde y lo copiaremos.

Para continuar con el registro debemos acceder al fichero oinkmaster.conf que es el fichero de configuración y buscar la línea:

```
# url = http://www.snort.org/pub-bin/oinkmaster.cgi/
```

```
/snortrules-version_descargada.tar.gz
```

Debemos editar este fichero eliminando la almohadilla del principio e incluyendo el código que obtuvimos en un primer momento después de *oinkmaster.cgi/* Quedando de la siguiente manera:

```
# url = http://www.snort.org/pub-bin/oinkmaster.cgi/
```

```
código_oink/snortrules-version_descargada.tar.gz
```

Ahora podemos descargar el paquete de reglas a partir de *oinkmaster* de la siguiente manera:

```
# perl oinkmaster.pl -C oinkmaster.conf -o ./
```

Una vez descargadas, las descomprimimos y las movemos al directorio de reglas que creamos anteriormente y creamos un enlace simbólico.

```
# mv *.rules /etc/snort/rules/
```

```
# ln -s /usr/local/bin/snort /usr/sbin/snort
```

Para poder trabajar mejor con *Snort* podemos crear un usuario y un grupo específicamente para ello, lo hacemos así:

```
# groupadd snort
```

```
# useradd -g snort snort
```

Ahora debemos asignar los permisos de la siguente forma:

```
# chown snort:snort /var/log/snort/
```

```
# touch /var/log/snort/alert
```

```
# chown snort:snort /var/log/snort/alert
```

```
# chmod 600 /var/log/snort/alert
```

```
# touch /etc/snort/rules/local.rules
```

Este punto es el momento en el que debemos configurar *Snort*. Para ello abrimos el fichero *snort.conf* situado en */etc/snort/* y buscamos las líneas:

```
var RULE_PATH ../rules
var PREPROC_RULE_PATH ../preproc_rules
```

que deberán ser sustituidas por:

```
var RULE_PATH /etc/snort/rules
var PREPROC_RULE_PATH /etc/snort/preproc_rules
```

En este mismo fichero también debemos buscar la línea:

```
# output database: log, mysql, user=root password=test dbname=db
host=localhost
```

y quitar la almohadilla del principio. Debemos cambiar el *user* por *snort* y el *password* por la contraseña que queramos utilizar.

También debemos buscar las líneas:

```
# include $PREPROC_RULE_PATH/preprocessor.rules
# include $PREPROC_RULE_PATH/decoder.rules
```

y eliminar las almohadillas de ambas.

Ahora podemos configurar una base de datos para *Sno*rt ejecutando el siguiente código dentro de MySQL:

```
CREATE DATABASE dbsnort;
GRANT CREATE, INSERT, SELECT, DELETE, UPDATE ON
     dbsnort.* TOsnort@LOCALHOST;
SET PASSWORD FOR snort@LOCALHOST=PASSWORD(‘password‘);
FLUSH PRIVILEGES;
exit
```

A partir de ahora ya podemos ejecutar Snort de la siguiente manera:

Sudo snort -c /etc/snort -i eth0

con -c le indicamos en que ruta se van a encontrar el fichero de configuración y con -i que interfaz de red debe escuchar. Si trabajamos con otra interfaz debemos cambiar esto.

9.2. Instalación de MuleStudio

Para poder trabajar con la aplicación necesitamos descargar la interfaz gráfica de Mule. Para ello debemos registrarnos en su web [13] y posteriormente descargar MuleStudio. Existe la posibilidad de descargar una versión diferente para cada sistema operativo. Se puede obtener más información sobre MuleStudio y descargarlo en su web oficial [14]

Una vez descargada el IDE sólo debemos descomprimirlo en caso de que estemos trabajando en Linux o ejecutar el fichero de instalación si trabajamos en Windows.

Para cargar la aplicación en MuleStudio sólo es necesario hacer click derecho en el explorador de paquetes en la parte izquierda de la pantalla. Seleccionar la opción *import* y escoger Mule Studio *generated Deployable Archive*.

Ahora le damos a siguiente y escogemos el archivo comprimido que vamos a cargar, en este caso nuestra aplicación. Por último le damos a finalizar y nuestro proyecto se habrá cargado correctamente en el IDE MuleStudio.

*

Capítulo 10

Manual del usuario

Vamos a ver en este capítulo un pequeño manual de cómo puede utilizar un usuario nuestra herramienta. Si bien esta herramienta puede ser usada por cualquier tipo de usuarios es recomendable tener algunos conocimientos de seguridad. Sobre todo la persona que va a recibir el email con la información de los ataques que se detecten, ya que es la encargada de actuar para la defensa del sistema que está siendo atacado.

10.1. Ejecución del software

Para poder ejecutar la aplicación es necesario haber realizado los pasos que hemos visto en el manual de instalación en el capítulo 9

Para poder utilizar esta aplicación en primer lugar debemos ejecutar *Snort*. Para ello podemos utilizar la siguiente orden:

```
snort -c /etc/snort -i eth0
```

En la sección 10.2 vamos a ver otras opciones que se pueden utilizar para ejecutarlo. En caso de que no estemos trabajando con la interfaz eth0 debemos cambiarla por la que se esté utilizando.

En segundo lugar, una vez que *Snort* está monitorizando el tráfico en la red y genera los ficheros de alertas, es el momento de lanzar la aplicación. Para ello debemos abrir MuleStudio con la aplicación y hacer click derecho sobre el flujo y en el menú de opciones que tenemos hacer click sobre *run application*.

En este momento nuestra aplicación ya se encuentra defendiendo el equipo y en el momento en el que se produzca alguna alarma referente a un ataque se mostrará por pantalla y el encargado de la seguridad del sistema recibirá un correo electrónico con dicho informe.

En la figura 10.1 podemos ver el resultado de la ejecución del software y el informe que devuelve por pantalla una vez que ha detectado un ataque. La aplicación enviará por email este informa al responsable de las seguridad del sistema

```
Generando informe...

Informe creado

Informe enviado

Ataque FLOOD en email detectado a las: 11:35:01.706839
 En la IP: 192.168.1.35
 Y en el puerto de correo: 110
 Del protocolo : POP3
 Proveniente de la IP: 192.168.1.33
 Y del puerto: 110
```

Figura 10.1: Informe generado

10.2. Opciones de Snort

A la hora de poner en marcha *Snort* podemos hacerlo de diferentes maneras, si bien hemos visto la forma con la que vamos a trabajar en este proyecto, pero vamos a mostrar cuáles son las opciones más interesantes y con las que el usuario puede intentar sacarle más partido.

La opción -c es la que vamos a utilizar para indicar dónde se encuentra toda la configuración relativa a las reglas 2.2, preprocesadores y demás. Debemos utilizar esta opción para detectar los intrusos en nuestra red.

La opción -i indica la interfaz que vamos a escuchar, podemos utilizar la orden *ifconfig* en Linux o *ipconfig* en sistemas Windows para conocer que interfaz de red estamos utilizando y hacer así que *Snort* monitorice el tráfico de esa red.

La opción -I es útil cuándo utilicemos varias interfaces, en este caso se va a añadir el nombre de la interfaz en la que se ha producido la alerta al fichero de alertas.

La opción -n seguida de un número sirve para que salgamos del programa en el momento en el que se reciban tantas alertas como se le indique.

La opción -p sirve para deshabilitar el modo promiscuo para monitorizar la red.

La opción -T nos sirve para realizar un test sobre la configuración de *Snort* y comprobar así si existe algún problema y detectar el fallo.

La opción -v vamos a utilizar *Snort* en modo *sniffer* y se van a mostrar por pantalla las cabeceras de los paquetes TCP/IP. Si añadimos la opción -d vamos a visualizar además los campos de datos que pasen por la interfaz indicada. Para ver más detalles de esta información debemos usar la opción -e mostrándonos en este caso las cabeceras a nivel de enlace.

La opción -V nos devuelve la versión de *Snort* que estemos utilizando.

La opción -w nos indica cuáles son las interfaces disponibles en este momento, pudiendo utilizar esta información para rastrear los paquetes que pasen por ella.

*

Capítulo 11

Manual del desarrollador

En este capítulo vamos a ver un un breve manual para que un posible desarrollador pueda seguir trabajando a partir de este proyecto.

11.1. Creación de nuevos patrones de eventos

Uno de los principales puntos que puede ser incrementado por un desarrollador puede ser el hecho de incluir nuevos patrones de eventos con el objetivo de detectar nuevos ataques. Para ello se deben realizar 3 pasos.

En primer lugar, el desarrollador debe crear un nuevo patrón utilizando el lenguaje EPL. Para conocer la sintaxis de este lenguaje y ver todas las opciones para el desarrollo de nuevos patrones véase la sección 2.5.1 o el manual oficial de Esper [56]. Se recomienda también la lectura de otras referencias recomendadas [45, 50, 5]

En segundo lugar el desarrollador debe crear un *listener* asociado a este patrón que sea capaz de tratar la información que reciba a partir de la identificación del ataque y generar un informe relativo a la alarma asociada a este ataque. Para ello puede ver la sección 5.6.5 dónde se explica como se realiza la implementación de estas clases.

Por último el desarrollador debe incluir en la configuración del motor de CEP Esper dos líneas referentes al nuevo patrón y su correspondiente *listener* de la siguiente manera:

```
NewPattern nuevo = new NewPattern(epService.getEPAdministrator());
nuevo.addListener(new NewPatternListener());
```

De esta manera se pueden crear nuevos patrones de eventos aumentando así el número de ataques capaz de reconocer la aplicación.

11.2. Componentes de Mule ESB

En caso de que el desarrollador desea implementar alguna nueva opción como la utilización de nuevos sensores o la publicación de la información en diferentes subscriptores, es necesario conocer cuáles son los componentes que nos ofrece Mule ESB. Debemos diferenciar diferentes tipos de componentes, que son:

- *Endpoints*
- *Scopes*
- *Components*
- *Transformers*
- *Filters*
- *Flow Control*
- *Cloud Conectors*

Gracias a los *endpoints* vamos a poder obtener información de un servicio, de un sensor o de una aplicación. En nuestro caso hemos utilizado un componente *File* que nos permite recibir información a partir de un fichero, pero también tenemos componentes relativos al correo electrónico como POP3 o SMTP. Podemos recibir información a partir de FTP, HTTP, TCP, UDP o incluso de bases de datos JDBC o Twitter.

También tenemos otro tipo de componente conocido como *scope* y sirven para indicar como se crean y manejan los objetos dentro de los contenedores de Mule. Debemos destacar el componente *Message Enricher* que es capaz de enriquecer un mensaje que fluya por Mule con la información que le queramos añadir.

Las herramientas de tipo *components* en Mule Studio se pueden diferenciar en 3 bloques, generales que permiten añadir funcionalidades como echo o ficheros de log. Otro bloque son los *scripts* que facilitan la integración del software como servicio y actualmente están disponibles para Groovy, Java, JavaScript, Python, Ruby y Script. Y el tercer bloque son los WebService componentes tanto para REST como para SOAP.

Mule Studio también nos proporciona una gran cantidad de transformadores, gracias a ellos podemos transformar los mensajes de un tipo a otro para poder procesarlos de la manera que mejor nos convenga. En nuestro caso hemos utilizado uno para pasar de *File* a *String* y luego a partir de un transformador que permite al usuario que lo implemente como quiera hemos desarrollado uno que transforme de *String* a *Event*.

Otro tipo de componentes son los filtros, éstos permiten filtrar la información en función a ciertas condiciones que impongamos. Existen filtros de operadores lógicos como *and*, *or* y *not*. Filtros de expresiones, de excepciones o algunos personalizables. Nosotros hemos utilizado uno de tipo *payload* que evalúa el tipo del mensaje y filtra según sea de un tipo determinado o de otro.

También existen los *flow control* que sirven para controlar o encaminar los mensajes a diferentes destinatarios. Tenemos algunos controladores de este tipo que permiten el envío de mensajes a todos los receptores de información o algunos que pueden filtrar estos receptores en función a ciertas condiciones impuestas por el desarrollador.

Por último tenemos los componentes de tipo *cloud conector* que sirven para facilitar la integración de Mule ESB con aplicaciones web. Actualmente se

puede utilizar estos componentes para integrar nuestra aplicación con Twitter, Magento, CMIS, Mongo DB, Salesforce y Twilio.

Todas estas son las herramientas de las que disponemos gracias a Mule Studio, para conocer más en profundidad alguna de ellas véase algunas referencias recomendadas [14, 13, 49]

Bibliografía

[1] *Ad-aware*. Http://www.lavasoft.com/.

[2] *Codehaus*. Http://www.codehaus.org/.

[3] *Cpal*. Http://opensource.org/licenses/cpal-1.0.

[4] *darkstat*. Http://unix4lyfe.org/darkstat/.

[5] *Esper*. Http://esper.codehaus.org/.

[6] *Ettercap*. Http://ettercap.sourceforge.net/.

[7] *Gantter*. Http://app.gantter.com.

[8] *Gpl*. Http://www.gnu.org/copyleft/gpl.html.

[9] *Grep*. Http://unixhelp.ed.ac.uk/CGI/man-cgi?grep.

[10] *Hupnet*. Http://hupnet.sourceforge.net/.

[11] *Kismet*. Http://ettercap.sourceforge.net/.

[12] *libpcap*. Http://sourceforge.net/projects/libpcap.

[13] *Mule esb*. Http://www.mulesoft.org/.

[14] *Mulestudio*. Http://www.mulesoft.org/documentation/display/MULE3STUDIO/.

[15] *Netreg*. Http://www.net.cmu.edu/netreg/.

[16] *Ngrep*. Http://ngrep.sourceforge.net/.

[17] *nmap*. Http://nmap.org/.

[18] *Nwatch*. Http://seclists.org/nmap-hackers/2001/41.

[19] *Opennac*. Http://sourceforge.net/projects/opennac/.

[20] *Packetfence*. Http://www.packetfence.org/home.html.

[21] *Snort*. Http://www.snort.org/.

[22] *Spybot*. Http://www.safer-networking.org/es/spybotsd/index.html.

[23] *Spyware blaster*. Http://www.javacoolsoftware.com/spywareblaster.html.

[24] *Stringtokenizer*. Http://www.webtutoriales.com/articulos/java-stringtokenizer-y-split.

[25] *Subversion*. Http://subversion.tigris.org/.

[26] *Syslog*. Http://www.syslog.org/.

[27] *Syslserve*. Http://www.syslserve.com/.

[28] *Tcpdump*. Http://www.tcpdump.org.

[29] *traffic-vis*. Http://www.mindrot.org/traffic-vis.html.

[30] *Ucase*. Http://www.uca.es/grupos-inv/TIC025.

[31] *Windump*. Http://www.winpcap.org/windump/.

[32] *winpcap*. Http://www.winpcap.org/.

[33] *Winsyslog*. Http://www.winsyslog.com/.

[34] *Wireshark*. Http://www.wireshark.org/.

[35] *yuml*. Http://yuml.me/.

[36] *Abc*, 2011. Http://www.abc.es/20110915/sociedad/abci-perdidas-cibercrimen-201109151628.html.

[37] Asaf Adi y Opher Etzion. *Amit - the situation manager. The VLDB Journal The International Journal on Very Large Data Bases*, 13(2):177–203, 2004. ISSN 1066-8888. doi:10.1007/s00778-003-0108-y.

[38] Agente Logic. *RulePoint - complex event processing software.* http://www.agentlogic.com/products/rulepoint.html, 2009.

[39] M. H Ali, C. Gerea, B. S Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Ananthanarayan, A. Kirilov, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Ying Li, V. Di Nicola, X. Wang, David Maier, S. Grell, O. Nano, y I. Santos. *Microsoft CEP server and online behavioral targeting. Proc. VLDB Endow.*, 2:1558–1561, agosto 2009. ISSN 2150-8097.

[40] Darko Anicic y Paul Allen. *Etalis - event-driven transaction logic inference system.* http://code.google.com/p/etalis/, 2010.

[41] Victor Ayllon y Juan Manuel Reina. *CEP/ESP: procesamiento y correlacion de gran cantidad de eventos en arquitecturas SOA.* En *IV Jornadas Cientifico-Tecnicas en Servicios Web y SOA*, págs. 97–110. Sevilla, octubre 2008. ISBN 978-84-691-6710-6.

[42] Brandeis University and Brown University and MIT. *The aurora project.* http://www.cs.brown.edu/research/aurora/, 2010.

[43] Brandeis University and Brown University and MIT. *The borealis project.* http://www.cs.brown.edu/research/borealis/public/, 2010.

[44] Brown University and Brandeis University and MIT. *Medusa: Scalable distributed stream processing.* http://nms.csail.mit.edu/projects/medusa/, 2010.

[45] Esper Codehaus. *Esper.* http://esper.codehaus.org, 2010.

[46] CollabNet Inc. *open-esb: IEP - open source complex event processing (CEP) and event stream processing (ESP) engine.* https://open-esb.dev.java.net/IEPSE.html, 2010.

[47] Cornell University. *Cayuga.* http://www.cs.cornell.edu/bigreddata/cayuga/, 2010.

[48] Jeff Davis. *Open Source SOA.* Manning Publications, mayo 2009. ISBN 978-1-933988-54-2.

[49] David Dossot y John D'Emic. *Mule in Action*. Manning Publications, 2010. ISBN 9781933988962.

[50] EsperTech Inc. *EsperTech - event stream intelligence*. http://www.espertech.com/, 2010.

[51] Opher Etzion y Peter Niblett. *Event Processing in Action*. Manning Publications, agosto 2010. ISBN 9781935182214.

[52] Event Zero. *Event zero - next generation monitoring solutions*. http://www.eventzero.com/, 2010.

[53] IBM. *Event-based middleware and solutions*. http://www.research.ibm.com/haifa/dept/services/soms_ebs.html, febrero 2007.

[54] IBM. *Stream computing - InfoSphere streams - software*. http://www-01.ibm.com/software/data/infosphere/streams/, noviembre 2010.

[55] IBM. *WebSphere business events version 7.0 - WebSphere business events - software*. http://www-01.ibm.com/software/integration/wbe/, noviembre 2010.

[56] EsperTech Inc. *Esper 4.0.0 - reference documentation*, 2009.

[57] JBoss Community. *Drools fusion*. http://www.jboss.org/drools/drools-fusion.html, 2010.

[58] Middleware Systems Research Group. *PADRES: a reliable Publish/-Subscribe middleware*. http://research.msrg.utoronto.ca/Padres/, 2010.

[59] Nastel Technologies Inc. *Autopilot m6 CEP - application performance management*. http://www.nastel.com/business-transaction-performance-reduce-risks-and-ensure-stability_99_86.html, 2010.

[60] Oracle. *Oracle complex event processing*. http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html, 2010.

[61] Adrian Paschke. *A semantic design pattern language for complex event processing*. En *Intelligent Event Processing, Papers from the 2009 AAAI Spring Symposium*, págs. 54–60. 2009.

[62] Progress Software Corporation. *Apama capital markets and complex event processing.* http://web.progress.com/es-es/apama/index.html, 2010.

[63] J.B. Puig, G.O. Bellot, y I.M. Bulo. *Procesamiento de eventos complejos en entornos SOA: caso de estudio para la deteccion temprana de epidemias.* 2011.

[64] RTM Realtime Monitoring GmbH. *RTM realtime monitoring GmbH.* http://www.realtime-monitoring.com/, 2010.

[65] ruleCore. *ruleCore - complex event processing server.* http://www.rulecore.com/, 2010.

[66] Sopera GmbH. *SOPERAï¿œ-ï¿œopen source SOA.* http://www.sopera.de/en/home/, 2010.

[67] Stanford University. *The stanford rapide project.* http://complexevents.com/stanford/rapide/, 2010.

[68] Stanford University. *Stanford stream data manager.* http://infolab.stanford.edu/stream/, 2010.

[69] Starview Technology. *Starview event servers.* http://www.starviewtechnology.com/starview-servers.html, 2010.

[70] StreamBase. *Complex event processing, event stream processing, StreamBase streaming platform.* http://www.streambase.com/, 2009.

[71] Sybase Inc. *Complex event processing (CEP) technology & Real-Time business process management.* http://www.sybase.com/products/financialservicessolutions/complex-event-processing, 2010.

[72] TIBCO Software Inc. *Service oriented architecture (SOA) software, business process management (BPM) software leader.* http://www.tibco.com/, 2010.

[73] Truviso. *Truviso web analytics software | scalable, real-time, Multi-Source web analytics tools.* http://www.truviso.com/, 2010.

[74] UC Berkeley. *The telegraph project.* http://telegraph.cs.berkeley.edu/, 2010.

[75] UC4. *Event processing - IC4 v8.* http://www.uc4.com/products-solutions/automation-engine/event-processing.html, 2010.

[76] University of Marburg. *PIPES - a public infrastructure for processing and exploring streams.* http://dbs.mathematik.uni-marburg.de/Home/Research/Projects/PIPES/, 2010.

[77] University of Massachusetts Amherst. *SASE.* http://sase.cs.umass.edu/, 2010.

[78] WestGlobal Ltd. *Vantify.* http://www.westglobal.com/, 2010.