COMUNICACIÓN ENTRE LOS COMPONENTES DE UN SISTEMA DE DETECCIÓN DE ATAQUES COORDINADOS

Memoria del proyecto de final de carrera correspondiente a los estudios de Ingeniería Superior en Informática presentado por Ignasi Barrera Caparrós y dirigido por Joan Borrell Viader.

Bellaterra, Febrero de 2007

El firmante, Joan Borrell Viader, profesor del Departament d'Enginyeria de la Informació i de les Comunicacions de la Universitat Autònoma de Barcelona
CERTIFICA:

Que la presente memoria ha sido realizada bajo su dirección por Ignasi Barrera Caparrós

Bellaterra, Septiembre de 2007

Firmado: Joan Borrell Viader

A mi familia, amigos, Chikipark y Coma-rugueros.

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Motivación

Cuando un atacante se hace con el control de una red informática, comprometiendo sus usuarios, equipos, datos o aplicaciones, la red y sus recursos pueden acabar siendo utilizados como parte activa de un ataque mayor como, por ejemplo, un escaneo de puertos coordinados, o una denegación de servicio distribuida contra terceras partes. Ambos tipos de ataques, tanto los coordinados como los distribuidos, se basan en la realización de acciones elementales, llevadas a cabo por un adversario malicioso que trata de violar la política de seguridad de un sistema remoto. La prevención de este tipo de ataques no es sencilla, ya que requiere una recogida de información global, que permita la generación de una vista al completo del sistema a proteger. Por ello, diferentes eventos deberán ser recolectados y puestos en común, en busca de conexiones sospechosas, iniciación de procesos anómalos, modificación de información confidencial, etcétera.

Es evidente la necesidad de una plataforma para la detección y prevención de este tipo de ataques. Dicha plataforma deberá recoger información de los diferentes elementos de protección de la red y ponerla en común mediante un proceso de correlación de alertas para construir los posibles escenarios de ataque. A través del desarrollo de una infraestructura de comunicaciones cooperativa basada en el paso de mensajes, los distintos componentes de la plataforma colaborarán en el proceso de detección y reparación de los ataques descubiertos. De esta forma, será posible tratar de prevenir la finalización de estos ataques, tan pronto como las primeras fases o los conjuntos de acciones elementales de estos ataques sean descubiertos a través del proceso de correlación.

Para que el proceso de detección, análisis y correlación de las alertas generadas por el sistema para la posterior toma de decisiones pueda llevarse a cabo de una manera eficiente, es necesario que tanto la información que se intercambia entre los diferentes elementos del sistema, como la manera y los protocolos mediante los que esta información

es transmitida, sean seguros, fiables y eficientes.

1.2. Objetivos

Los objetivos del proyecto consisten en analizar y realizar una primera implementación del subsistema de comunicaciones de la plataforma para la detección y prevención de ataques coordinados que se está desarrollando, y comprobar qué herramientas y tecnologías pueden ser las más adecuadas. Podrían resumirse en los siguientes puntos:

- Analizar y estudiar el formato de alertas IDMEF [2] (Intrusion Detection Message Exchange Format) para su posterior implantación como el formato a utilizar para el intercambio de información entre los elementos de la plataforma.

- Estudiar el modelo de comunicación publicador/suscriptor y adoptar un framework de comunicaciones que implemente este paradigma.

- Diseñar publicadores y suscriptores capaces de recoger las diferentes alertas generadas por el IDS (Intrusion Detection System) y publicarlas en la red de comunicaciones.

- Construcción de un entorno completo que simule el comportamiento de una posible plataforma de detección de ataques coordinados.

1.3. Contenidos de la memoria

El trabajo se ha organizado de la siguiente manera:

- **Conocimientos previos**
  En este capítulo se expondrán los conceptos teóricos necesarios para comprender el trabajo realizado y las decisiones de diseño adoptadas, así como la presentación de las diferentes tecnologías utilizadas en la implementación.

- **Análisis**
  En este apartado se analizan los problemas que pueden surgir en la realización de este proyecto, y se discutirán los pros y los contras de las diferentes posibilidades tanto de diseño, como de la tecnología utilizada.

- **Herramientas utilizadas**
  En este capítulo se presentarán las herramientas utilizadas, se mostrarán sus características y se detallará su funcionamiento. Se expondrán también las principales ventajas que aportan y que nos han hecho decidirnos por ellas para implementar la plataforma de detección de ataques.

- **Diseño e implementación**
  Aquí se expondrán las decisiones de diseño adoptadas para nuestra infraestructura de comunicaciones. Se hará una descripción de los diferentes componentes que formarán nuestra plataforma y se dará una visión global del sistema. También se expondrán el proceso y los detalles de la implementación, los problemas encontrados y las soluciones adoptadas.

- **Evaluación de la plataforma**
  En este capítulo se mostrará el estudio realizado para la evaluación de la plataforma y sus elementos. En él se analiza el consumo de CPU, de memoria, los tiempos de publicación, de suscripción, de parseo de alertas y de todos aquellos parámetros que pueden condicionar el rendimiento del sistema.

Capítulo 2

Conocimientos previos

Entendemos por ataque coordinado la combinación de acciones llevadas a cabo por uno o varios adversarios maliciosos cuyo objetivo es violar la política de seguridad del sistema informático objetivo. Cuando esto sucede, los recursos y elementos de la propia red pueden convertirse en parte activa del ataque coordinado, por ejemplo si el ataque empieza con una intrusión exitosa en alguno de los elementos de la red.

Para prevenir este tipo de acciones, es necesaria una monitorización activa de los nodos de la red, así como una visión global del sistema completo y de su estado. Para ello se necesita recoger información relacionada con lo que ocurre en el sistema y procesarla para una posterior toma de decisiones.

2.1. Introducción a los IDS

Un Sistema de Detección de Intrusiones es un programa o conjunto de programas utilizados para detectar comportamientos que puedan comprometer la política de seguridad de un computador o una red de computadores, que no pueden ser detectados por elementos más sencillos, como por ejemplo los cortafuegos. La manera de detectar estos comportamientos dependerá en gran medida de la información que esté monitorizando el IDS, y de cómo éste está estructurado.

2.1.1. Tipos de IDS

- **HIDS** (*Host-based IDS*): Es un IDS que monitoriza una única máquina. Monitoriza las modificaciones en el sistema de ficheros (binarios, ficheros de contraseñas, capacidad, listas de control de acceso, bases de datos, etc), llamadas al sistema, ficheros de log y otras actividades que ocurren en la propia máquina.

- **NIDS** (*Network-based IDS*): Es un IDS que monitoriza la información que llega

desde el exterior del sistema. Analiza el tráfico de red, en busca de paquetes fraudulentos, mal formados, patrones de cadenas de desbordamiento de buffer, escaneos de puertos, etc. Un ejemplo de IDS basado en red es Snort [6].

- **PIDS** (*Protocol-based IDS*): Consiste en un IDS que se sitúa en frente de un servidor en concreto y monitoriza y analiza la comunicación entre el servidor y el cliente conectado. Para un servidor web, por ejemplo, el IDS analizaría el protocolo HTTP en busca de posibles malformaciones.

- **AIDS** (*Application Protocol-based IDS*): Este IDS se situaría en un grupo de servidores monitorizando y analizando los protocolos de aplicación específicos. Por ejemplo, en un servidor web con base de datos, este IDS monitorizaría el protocolo SQL específico de la lógica de negocio mientras interactúa con la base de datos.

- **Híbridos**: Estos IDS son la combinación varios de los anteriores. Un ejemplo de IDS híbrido es Prelude [5].

También pueden clasificarse en sistemas *pasivos* o *reactivos*. Llamaremos IDS pasivo a aquel que cuando una alerta es generada ante un posible escenario de ataque, almacena dicha información para un proceso posterior; y llamaremos IDS reactivo a aquel que responde a la actividad sospechosa tratando de contrarrestarla.

2.1.2. Componentes

Los Sistemas de Detección de Intrusiones anteriormente mencionados, utilizan diferentes componentes para obtener la información necesaria, procesarla, construir posibles escenarios de ataque y tomar las medidas de prevención necesarias para evitar que un nodo de la red se vea comprometido por el posible atacante.

Los principales componentes de un Sistema de Prevención de Ataques son los siguientes:

- **Sensores:** Son los elementos encargados de recoger la información del medio y reportarla en forma de eventos al IDS para que pueda ser analizada y correlacionada con la información obtenida por otros sensores. Podríamos clasificarlos en sensores (*Host-based*) que monitorean la propia máquina (ficheros de log, ficheros de contraseñas, etcétera) y que generan eventos cuando se cumplen o incumplen ciertas políticas, y en sensores (*Network-based*) que analizan la información que proviene del exterior.

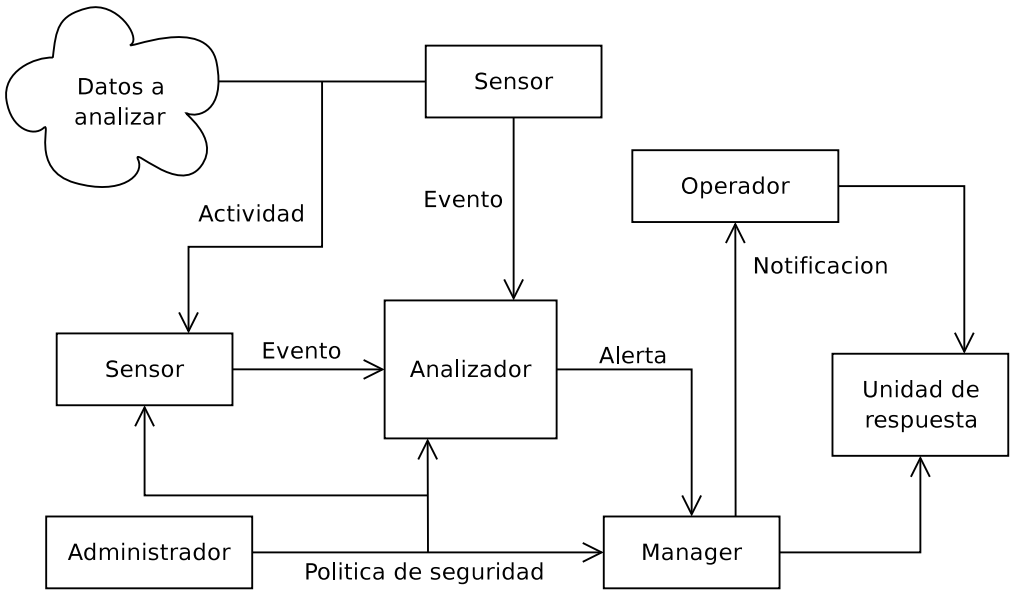


Figura 2.1: Esquema general de un IDS

- **Analizadores:** Reciben los eventos generados por los sensores y los analizan, en busca de comportamientos anómalos, huellas de ataques conocidos, uso erróneo de los recursos, y en general de cualquier indicio que pueda identificar una acción como parte de un ataque.

- **Managers:** En la detección de ataques coordinados no es suficiente la detección de una acción anómala para afirmar que se está produciendo un ataque contra el sistema. Cada alerta puede carecer de significado si se procesa de forma individual, pero un conjunto de ellas pueden constituir un ataque. Los *Managers* de los IDS reciben el conjunto de alertas generadas y llevan a cabo los procesos de correlación necesarios para generar los posibles escenarios de ataque.

- **Unidades de Respuesta:** Son elementos propios de los IDS reactivos. Una vez construidos los escenarios de ataque, es posible que sea preciso realizar ciertas acciones como cerrar puertos, aplicar reglas a un cortafuegos, cerrar ficheros, etc. Las *Unidades de Respuesta* son las encargadas de realizar dichas acciones en base a la información facilitada por los *Managers* del IDS.

2.1.3. Arquitectura

Existen diferentes arquitecturas mediante las que se puede diseñar un Sistema de Detección de Intrusiones.

Una arquitectura centralizada, sería aquella en la que un único elemento de la plataforma se encargara de procesar la información recibida de los diferentes sensores de que consta el sistema.

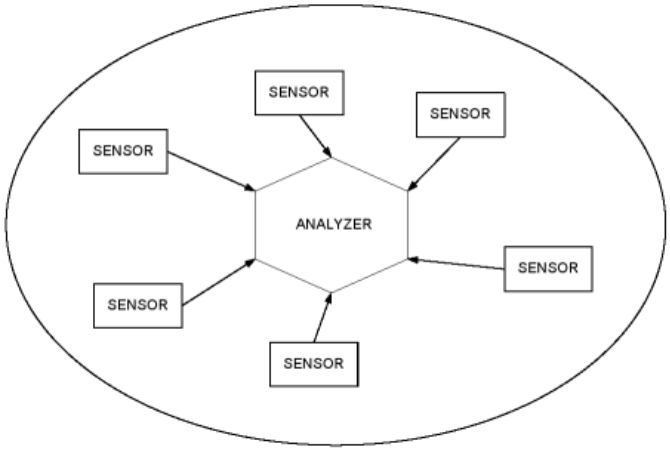


Figura 2.2: Correlación de eventos centralizada

Este esquema tiene como principal ventaja la sencillez de la implementación. Todo el trabajo se realiza en un único punto y no es necesario interactuar con otros elementos del sistema para construir los escenarios de ataque. Asimismo, presenta numerosas desventajas: no es escalable, puede convertirse fácilmente en un cuello de botella comprometiendo severamente el rendimiento del sistema; todo el trabajo se concentra en un único punto que será con toda seguridad el objetivo de muchos ataques, por ejemplo de Denegación de Servicio (DoS), que podrían provocar una caída o saturación de dicho elemento que supondría prácticamente la anulación del IDS.

Una arquitectura jerárquica, es aquella en que tenemos diferentes analizadores distribuidos en distintos dominios, que reciben los eventos y notificaciones de los sensores locales. Estos analizadores, pueden realizar un proceso local de las alertas, y enviar a un analizador padre los resultados, para que genere los posibles escenarios de ataque a partir de la visión global del sistema, obtenida de los analizadores de los diferentes dominios.

Esta aproximación resuelve en parte el problema de la centralización del trabajo y la mala escalabilidad del modelo centralizado. Ya no se tiene un solo nodo del sistema procesando el trabajo sino que se divide la red en diferentes dominios en los que se realiza un procesamiento local. Es más complicado comprometer al sistema dejando un nodo inutilizado, pero todavía existe el problema del cuello de botella en el nodo padre. Este nodo sigue siendo un potencial objetivo de ataques y su rendimiento puede condicionar el rendimiento de todo el sistema.

Las arquitecturas distribuidas son aquellas en las que no existe ningún nodo raíz sobre el que resida el trabajo principal. Los nodos de la red del Sistema de Detección de Intrusiones deben colaborar unos con otros para intercambiar la información necesaria para la construcción de los posibles escenarios de ataque. La complejidad del diseño se incrementa considerablemente, ya que un nodo no solamente ha de ser capaz de realizar el trabajo local, sino que ha de poder enviar la información a una serie de nodos y a su vez recibir y tomar en consideración la información recibida de los nodos vecinos. Este tipo de estructura, es mucho más escalable y pueden definirse células más complejas, es completamente descentralizado, por lo que resolvemos el problema del cuello de botella al no tener un único elemento del sistema soportando la mayor parte de la carga de trabajo, y desaparece la dependencia de un único nodo para el correcto funcionamiento del sistema.

Es evidente, que para una plataforma para la prevención de ataques coordinados la arquitectura que más se amolda a las necesidades es la distribuida. En ella cada nodo juega un papel igual de importante que el resto, y aportará la información necesaria para construir, junto con la información recibida, una vista global del estado del sistema.

La dificultad de este tipo de arquitectura respecto a las anteriores, reside en la implementación, y en la adopción de un modelo de comunicaciones y de un modelo para la información que se va a compartir, que cumpla todas las necesidades, sea eficiente y seguro. Existen diferentes paradigmas que solucionan los diferentes problemas de comunicación en redes distribuidas, siendo unos más adecuados que otros para un IDS. En este proyecto hemos optado por un modelo de comunicación basado en el paso de mensajes utilizando el sistema **publicador/suscriptor**. Más adelante, discutiremos los diferentes modelos de comunicación que podrían plantearse, y las ventajas e inconvenientes que presenta cada uno para comprender el por qué de la adopción de este modelo.

2.2. El formato de alertas IDMEF

Hemos visto que en los Sistemas de Detección de Ataques Distribuidos, un aspecto muy importante es el intercambio de información entre los diferentes nodos. Los datos intercambiados han de ser completos y han de contener toda la información necesaria para que los receptores puedan utilizarlos.

Cuando un analizador del Sistema de Detección de Intrusiones notifica un evento en forma de alerta, éste se encargará de rellenarla con la información relacionada tanto con el propio evento, como con los componentes del sistema de detección y con el entorno donde se encuentra instalado. La inclusión de esta información en el interior de la alerta

supondrá una mejora semántica y permitirá facilitar el tratamiento de un evento de una forma más eficiente.

Si nos encontramos ante un analizador basado en anomalías, por ejemplo, la alerta contendrá las métricas utilizadas por el analizador, las variables instanciadas, una posible clasificación de la anomalía detectada, etc. Si dicho analizador se encarga de analizar eventos de un sensor de red, la alerta contendrá información referente al tráfico de red asociado, las direcciones de origen y destino, el protocolo utilizado, etc. Si el evento proviene de un sensor basado en equipo, el analizador incluirá la información asociada al proceso que generó dicho evento.

En un contexto de análisis cooperativo como el nuestro, puesto que las alertas provendrán de distintos componentes que cooperan en el proceso de detección y correlación, es necesaria la utilización de un formato de alertas común que permita un diálogo entre los distintos elementos del sistema. Tal formato deberá ser lo suficientemente expresivo como para permitir la transmisión de las diferentes informaciones generadas a medida que se vayan analizando los eventos reportados por los sensores. Por otro lado, y puesto que el tratamiento de alertas generadas se realizará en cadena, será necesario que dicho formato sea extensible. Esta propiedad nos permitirá la adición de nueva información a las alertas resultantes de un proceso de detección dinámico.

Aunque la mayoría de productos propietarios basan el formato de sus alertas en soluciones propias, la creación de un formato común de alertas ha sido propuesto por la comunidad de detección de intrusos. EL *Common Intrusion Specification Language*, por ejemplo, fue propuesto unos años atrás para permitir que los componentes del *Common Intrusion Detection Framework* pudiesen intercambiar la información de una forma semánticamente bien definida. Otros lenguajes con el mismo propósito son el *Intrusion Detection Message Exchange Fotmat* (IDMEF), propuesto por el *Intrusion Detection Exchange Format Working Group* (IDWG) del IETF, y el *Security Device Event Exchange* (SDEE), propuesto por el *Intrusion Detection Consortium* (IDSC).

De entre estos lenguajes, hemos escogido el estudio del formato IDMEF para la comunicación entre los componentes del Sistema de Detección de Intrusiones. En primer lugar, este formato permite una excelente especificación de las alertas generadas a partir de la información obtenida de los sensores tanto si son de red o basados en equipo. Aparte, es un formato extensible y ofrece la posibilidad de incorporar información adicional en el interior de las alertas. Por último, existe un gran número de herramientas e implementaciones de IDS actuales basados en el formato IDMEF, lo que facilitará la búsqueda de información durante la fase de desarrollo del proyecto.

El formato IDMEF es la especificación de un formato que pretende ser un estándar

(por el momento su especificación reside en un borrador de Internet que debe ser actualizado cada seis meses) para el intercambio de alertas entre componentes de distintos Sistemas de Detección. Aunque la idea inicial de IDMEF es ser un transporte de datos específico entre los componentes de un Sistema de Detección, su utilización se puede expandir a escenarios más generales. Por ejemplo, IDMEF puede ser utilizado para la comunicación entre los distintos elementos de un mismo componente, el almacenamientos de alertas en una base de datos local o remota, etc. Esta propiedad es interesante para nuestro proyecto ya que nos permite la concepción de una arquitectura modular en la que la inclusión de nuevos elementos para el tratamiento de eventos no es un problema. Por lo tanto, la utilización de IDMEF dentro de nuestra propuesta de proyecto nos permite el intercambio de información y facilita la comunicación entre los elementos que participan en los distintos eslabones de la cadena de análisis, detección y almacenamiento de la información.

2.2.1. Modelo de datos del formato IDMEF

El modelo de datos del formato IDMEF [2] es orientado a objetos, y por tanto es extensible a través de la derivación de clases o de la agregación de clases nuevas. Así pues, un componente del sistema capaz de tratar alertas instanciadas a partir de un modelo IDMEF no extendido será también capaz de tratar alertas instanciadas a partir de un modelo IDMEF extendido. Simplemente, el componente descartará la información adicional del modelo extendido.

Aunque las primeras versiones del modelo IDMEF vienen especificadas por un DTD (*Document Type Definition*) de XML, la versión definitiva de IDMEF vendrá definida por un *Schema* de XML, en lugar de un DTD. Este cambio se debe a que un DTD de XML no soporta la especificación de herencia que el modelo IDMEF pretende (relaciones de tipo *kind-of*, por ejemplo). Aunque este inconveniente ha sido solventado hasta el momento mediante relaciones de agregado (con una mínima pérdida de funcionalidad), la utilización de *Schemas* de XML proporciona el soporte necesario para las relaciones de herencia, así como otras características interesantes. El motivo de no utilizar por el momento los *Schemas* es que las recomendaciones del *W3C* (The World Wide Web Consortium) aún no están finalizadas.

La clase base de IDMEF es *IDMEF-Message* que a su vez deriva en dos subclases: *Hearbeat* y *Alert*. La primera, *Heartbeat*, está pensada para dar a conocer el buen funcionamiento de los distintos componentes del sistema a través de mensajes IDMEF. La utilización de esta clase *Heartbeat* no tiene especial interés para el desarrollo de nuestro

marco de trabajo cooperativo. Por ello, su presentación y utilización quedarán fuera del presente documento.

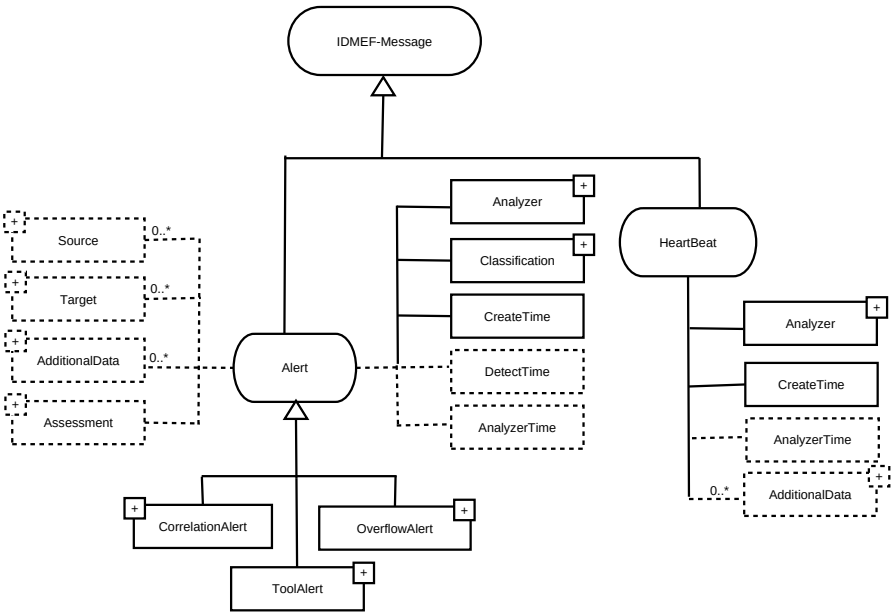


Figura 2.3: Estructura de la clase *IDMEF-Message*

La segunda clase derivada de *IDMEF-Message*, *Alert*, es la instancia de la alerta lanzada en el interior del sistema de detección que nos interesa comunicar a través de IDMEF. Las clases agregadas a la clase *Alert*, y que constituyen sus atributos son las siguientes:

- *Analyzer*: Esta clase recoge la información relativa al analizador del sistema que ha generado la alerta (su dirección IP, su identificador, su localización, etc.). También permite destacar si la alerta ha sido ya tratada por algún otro componente del sistema. Para ello, una instancia de la clase *Analyzer* puede contener en su interior referencias a la propia clase. De esta forma será posible añadir nuevas instancias de la propia clase en cada paso que la alerta sea tratada (dentro de la cadena de análisis general del sistema de detección) y poder así trazar el camino realizado por la alerta. Es importante remarcar que una misma instancia de la clase *Alert* posee exactamente una única instancia de la clase *Analyzer*.

- *CreateTime*: Una instancia de esta clase precisa la marca de tiempo en la cual la alerta ha sido creada. Esta marca de tiempo corresponderá al momento en el que los datos incluidos en la alerta fueron generados y tratados por el analizador.

- *DetectTime*: Una instancia eventual de esta clase precisa la marca de tiempo en la cual el evento original que da lugar a la creación de la alerta ha sido detectado. Esta

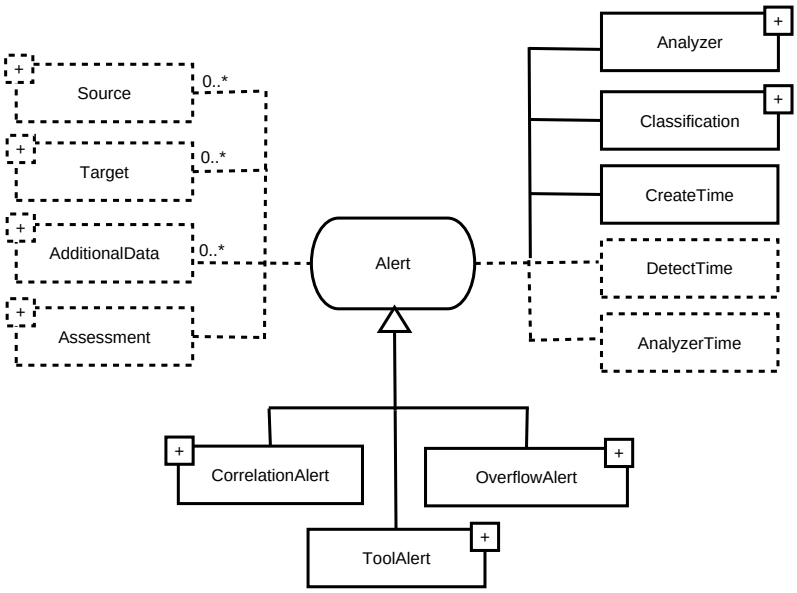


Figura 2.4: Estructura de la clase *Alert*

marca de tiempo es opcional y no tiene por qué coincidir con el valor de la marca de tiempo en el momento de creación de la alerta. De hecho, entre el momento de detección de la alerta y su posterior creación por parte del analizador, existe un retardo que se puede agravar en función de diferentes factores (arquitectura, sistema operativo, etc.).

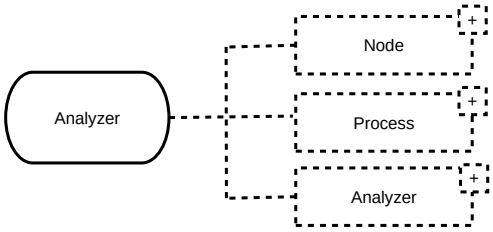


Figura 2.5: Estructura de la clase *Analyzer*

- *AnalyzerTime*: una instancia eventual de esta clase precisa la marca de tiempo en el que el presente analizador acaba de tratar la alerta. No siempre esta marca de tiempo coincidirá con el momento de creación de la alerta.

- *Source*: Es una de las *core-classes* del formato IDMEF. Encapsula la información relativa a la entidad que aparece como origen del evento asociado a la creación de la alerta. Parte de esta información puede ser relativa al usuario de la entidad de origen, el dispositivo en cuestión (tipo, dirección, etc.), el proceso de origen y los servicios de red concernientes. La instanciación de esta clase en una alerta puede ser múltiple, aunque como muestra el diagrama de clases, es opcional.

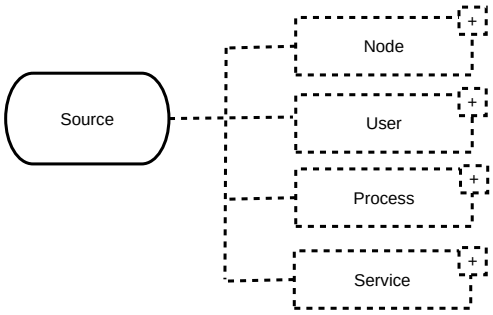


Figura 2.6: Estructura de la clase *Source*

- *Target*: Es otra de las *core-classes* del formato IDMEF. Encapsula la información relativa a la entidad que aparece como objetivo del evento asociado a la creación de la alerta. Esta clase recoge la misma información que la clase *Source* y añade también información sobre los ficheros relacionados con el evento.

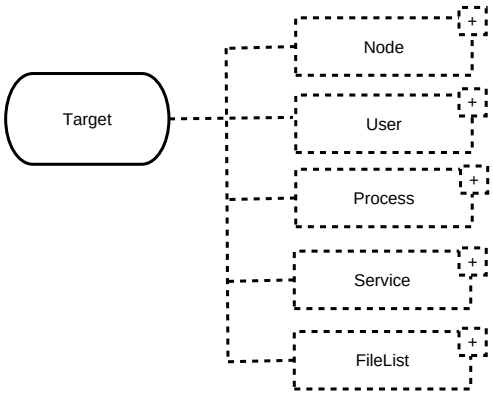


Figura 2.7: Estructura de la clase *Target*

- *Classification*: Contiene el nombre del evento asociado a la alerta. Este nombre es necesario para el resto de elementos que tratarán la alerta, con el objetivo de clasificarla y deducir las propiedades que irán ligadas a ella. Será necesario, por tanto, que estos elementos tengan el conocimiento necesario para identificar y comprender el evento asociado. Si es posible, se añadirá, a modo de información complementaria, una URL que apuntará a una pequeña explicación acerca de la alerta detectada. Este último parámetro es opcional.

- *Assessment*: Esta clase encapsula la información relativa a la detección del evento que originó la alerta. Una instancia eventual de esta clase informará sobre el impacto del evento sobre el objetivo (siempre que el sistema tenga el conocimiento suficiente como para evaluarlo), el grado de confianza sobre el diagnóstico dado

por el sistema de detección y las posibles acciones que se podrían ejecutar en forma de contramedidas al evento que se ha producido.

- *ToolAlert*: Encapsula la información relativa a la herramienta utilizada por la entidad que provocó el evento (por ejemplo, un atacante). Mediante esta información será posible generar clusters o conjuntos de alertas asociadas a la ocurrencia del evento creado por la utilización de dicha herramienta.

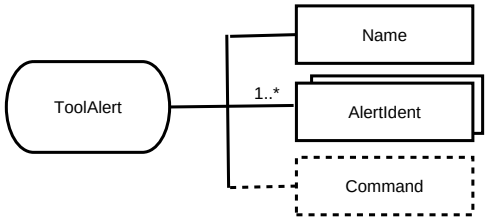


Figura 2.8: Estructura de la clase *ToolAlert*

- *CorrelationAlert*: Permite precisar un conjunto de identificadores de alertas IDMEF relacionadas entre ellas. Esta información permitirá también precisar qué técnica se ha utilizado para el reagrupamiento de tales identificadores.

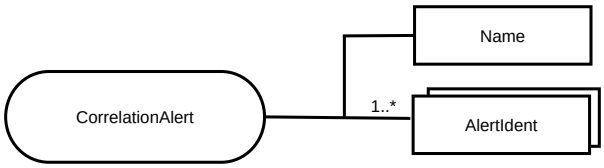


Figura 2.9: Estructura de la clase *CorrelationAlert*

- *OverflowAlert*: Permite precisar los datos relativos a un evento debido a un ataque de tipo *buffer overflow*. La información contenida en esta clase ayudará también a indicar el proceso explotado y el *shell code* utilizado para efectuar el ataque.

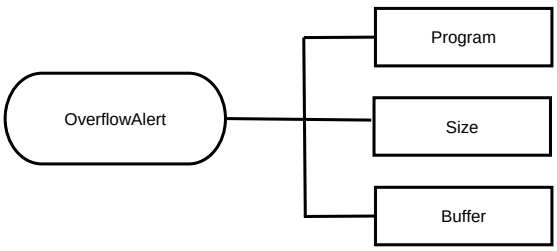


Figura 2.10: Estructura de la clase *OverflowAlert*

- *AdditionalData*: Esta clase permite la inclusión dentro de una alerta de todos aquellos datos no representables dentro del modelo IDMEF estándar. De esta forma, el

modelo puede ser extendido de manera sencilla, sin necesidad de modificar el DTD o *Schema* de IDMEF.

2.3. Modelo de comunicación publicador/suscriptor

El modelo publicador/suscriptor está especialmente pensado para situaciones de comunicación en grupo, es decir, situaciones donde un mensaje (o notificación) es enviado por una única entidad y es requerido por, o distribuido para, múltiples entidades. Generalmente es utilizado para la diseminación de información de manera cómoda y eficiente entre distintos elementos de un mismo grupo. Al contrario que en el modelo de comunicación *multicast*, los clientes del modelo publicador/suscriptor tienen la posibilidad de describir los eventos en los que están interesados de una forma más específica (por ejemplo, basándose en el contenido o en las cabeceras de la notificación).

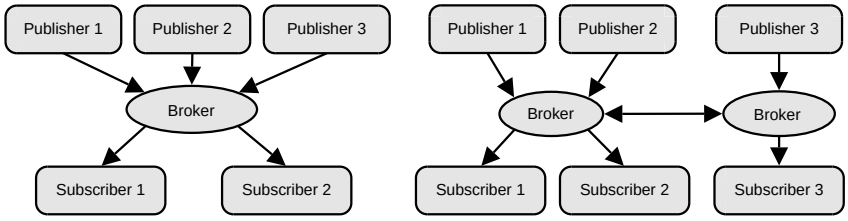
2.3.1. Sistemas publicador/suscriptor

Un sistema publicador/suscriptor consiste en, al menos, un encaminador (o *broker*) que redirija las notificaciones publicadas por parte de los clientes del sistema hacia otros clientes que han mostrado su interés en recibir dichas notificaciones. Por razones de escalabilidad, es normal la implementación de una red distribuida de encaminadores, lo que conforma el *servicio de notificaciones*, es decir, el servicio que conforma el solapamiento de los distintos encaminadores del sistema. Este servicio proporcionará una infraestructura distribuida para el encaminamiento de notificaciones (incluyendo además las notificaciones necesarias para la gestión y mantenimiento del propio servicio de encaminamiento).

La entrega de las notificaciones podrá realizarse, por tanto, de manera síncrona o asíncrona. Los clientes pueden publicar notificaciones, o suscribirse a los filtros necesarios que vincularán al cliente con aquellas notificaciones entregadas a los encaminadores del sistema que cumplan las condiciones de dichos filtros En cuanto un encaminador recibe una nueva notificación, se encargará de comprobar y verificar sus suscripciones locales y, en caso de encontrar coincidencias, de entregarla al cliente que lo ha solicitado. Adicionalmente, el encaminador se encargará de reenviar también las nuevas notificaciones a otros encaminadores que convivan en el mismo grupo (es decir, aquellos que estén configurados para actuar como repetidores del encaminador actual).

Un ejemplo de sistema publicador/suscriptor mono-encaminador es el mostrado en la figura 2.11(a). Aquí cinco clientes están relacionados con un único encaminador de

la manera siguiente. Tres clientes están publicando notificaciones y otros dos clientes están suscritos a un subconjunto de las notificaciones publicadas en ese encaminador. Los suscriptores pueden decidir suscribirse o cancelar parte o la totalidad de sus suscripciones de manera independiente y autónoma. El encaminador se encargará de hacer llegar las notificaciones recibidas a los clientes que así lo deseen, garantizando de esta manera que cada notificación es entregada a todos los suscriptores interesados.



(a) Entorno mono-encaminador (b) Entorno multi-encaminador

Figura 2.11: Dos ejemplos de sistemas publicador/suscriptor

Este sencillo ejemplo puede ser extendido mediante la utilización de múltiples encaminadores configurados en forma de *cluster* (figura 2.11(b)), de manera que puedan intercambiar los mensajes recibidos, y encaminarlos hacia su destinatario final. Este nuevo diseño permite que clientes suscritos a uno de los encaminadores del sistema reciban mensajes que han sido publicados sobre otro encaminador, haciendo transparente de cara al cliente final el camino necesario para obtener estos mensajes.

La puesta en funcionamiento de este esquema es transparente para el desarrollador de aplicaciones publicador/suscriptor, ya que será el administrador del sistema quien deberá realizar la configuración del cluster. De esta manera, una aplicación podrá tener un enfoque centralizado o descentralizado, según la configuración del conjunto de encaminadores. Los suscriptores deberán únicamente realizar la petición de información a recibir, pudiendo elegir entre distintas metodologías, según el sistema publicador/suscriptor seleccionado. Dos de las metodologías más comunes son, respectivamente, la suscripción basada en temas (*topic-based subscription*) y la suscripción basada en contenidos (*content-based subscription*).

Una metodología de suscripción basada en temas es más sencilla de manejar en comparación a una tecnología de suscripción basada en contenido. Aquí, los suscriptores especifican su interés en un tema, y recibirán todos los mensajes publicados sobre este tema. Uno de los mecanismos generalmente utilizados para la combinación de mensajes suscritos y publicados es la utilización de un esquema basado en canales, de manera que una suscripción será combinada con una publicación en el sistema, si dicha suscripción coincide exactamente con la temática del canal donde dicho mensaje ha sido publicado.

Otros mecanismos más completos, como la gestión de temas en forma jerárquica, pueden ser también aplicados. En este segundo caso, las suscripciones podrán ser combinadas con la temática de un canal, o con aquellos subconjuntos de mensajes que deriven de dicho canal.

Por último, las suscripciones basadas en contenido permiten la realización de suscripciones de manera más sofisticada. Por el contrario, el coste en el rendimiento del sistema es mayor que la solución anterior, en especial a la hora de realizar las tareas de encaminamiento de los mensajes. Aquí, una suscripción puede ser formulada de manera más precisa, basándose en combinaciones realizadas sobre el contenido del mensaje a recibir, mediante la utilización de lenguajes basados en expresiones regulares o consultas que pueden llegar a ser extraordinariamente complejas.

2.4. Resumen

En este capítulo hemos dado una visión global de lo que es un Sistema de Detección de Intrusiones, de la tarea que realizan, y de los componentes necesarios para llevar a cabo las diferentes etapas del proceso de detección. Hemos presentado también el formato IDMEF (*Intrusion Detection Message Exchange Format*) como el mecanismo escogido para el intercambio y almacenamiento de la información transmitida entre los diferentes elementos del sistema, y hemos dado una primera visión de un sistema de comunicaciones orientado a grupos y basado en el paso de mensajes, que es el que adoptaremos como subsistema de comunicación de nuestra plataforma de Detección de Ataques Coordinados.

Capítulo 3

Análisis

3.1. Punto de partida

El objetivo principal de este proyecto es implementar un subsistema de comunicaciones basado en el paradigma publicador/suscriptor para el intercambio de información entre los diferentes componentes del Sistema de Detección de Intrusiones. La mayoría de soluciones para la detección y prevención de intrusiones que podemos encontrar hoy en día, utilizan métodos centralizados o jerárquicos, cuyo rendimiento, escalabilidad y fiabilidad puede verse incrementado con el uso de un sistema completamente descentralizado como el sistema publicador/suscriptor.

Antes de atacar el problema directamente, es necesario comprender las diferentes tecnologías de comunicación que podrían ser útiles para nuestro caso y decidir cuál es la más adecuada, así como asimilar los conceptos más importantes para el posterior diseño y desarrollo de la plataforma. No se pretende diseñar una nueva plataforma que implemente el sistema publicador/suscriptor, sino la adopción de una ya existente que cubra las necesidades de nuestro sistema de detección de ataques.

3.2. Intercambio de mensajes

Como hemos comentado, las soluciones para la comunicación de componentes en plataformas de gestión de ataques son, mayoritariamente, arquitecturas cliente/servidor que implementan esquemas de comunicación centralizados o jerárquicos. Estas soluciones presentan problemas relacionados con la saturación del servicio de comunicación por parte de analizadores centrales o del nivel superior de las jerarquías. Sistemas centralizados como DIDS y NADIR, que basan la correlación de la información en elementos dedicados, presentan fuertes restricciones en cuanto a escalabilidad y eficiencia. Éstos se encargan de almacenar toda la información recogida por el sistema en un punto central

para poder proceder a la ejecución del proceso de detección. Esquemas jerárquicos, como los utilizados por GrIDS y NetSTAT continúan utilizando nodos dedicados que actúan como puntos centrales de recogida de información. Aunque estos esquemas mejoran la escalabilidad respecto a esquemas centralizados, siguen presentando deficiencias similares y son vulnerables a problemas de saturación de servicio por parte de los nodos superiores de la jerarquía.

En contra de estos esquemas tradicionales, la mayor parte de propuestas alternativas tratan de eliminar la necesidad de utilizar elementos dedicados. La idea de distribuir el proceso de detección mediante la colaboración de distintos componentes de la plataforma de detección presenta grandes mejoras respecto a esquemas centralizados y jerárquicos. La principal ventaja de este nuevo diseño es la inexistencia de puntos de fallo aislados o cuellos de botella en el sistema.

Algunos diseños basados en paso de mensajes como, por ejemplo, CSm y Quicksand, tratan de eliminar esta necesidad de elementos centrales o jerárquicos mediante el uso de una arquitectura P2P (*Peer-to-Peer*). Así, en lugar de disponer de estaciones centrales o jerárquicas, plantean la utilización de entidades de trabajo distribuidas por los diferentes elementos a vigilar. Tales entidades tratan de realizar tareas equivalentes de correlación mediante esquemas cooperativos. Para ello, compartirán la información de sus respectivos procesos de detección mediante la utilización de una infraestructura de comunicación común que permitirá la realización de un algoritmo de correlación descentralizado.

Estos diseños, parecen apuntar en la buena dirección para la implementación de arquitecturas descentralizadas para la gestión de ataques e intrusiones. Sin embargo, las propuestas anteriores presentan un diseño muy limitado y sufren algunas carencias. En la mayor parte de esquemas alternativos estudiados, cada elemento requeriría un conocimiento global del sistema, de manera que todos los elementos han de estar conectados entre sí, y mantener de forma local la matriz de conexiones necesaria para poder comunicar la información con el resto. Esto hace que el rendimiento de dicho sistema sea extremadamente costoso de controlar y mantener, presentando una escalabilidad mínima. Además, cada elementos del sistema deberá realizar el trabajo propio dentro de la cadena del proceso de detección, además de comunicarse con el resto de nodos. La implementación de esta comunicación puede compleja en mayor o menor grado dependiendo de la estructura de la red; en todo caso la complejidad del diseño, implementación y mantenimiento de este tipo de nodos se eleva considerablemente.

Otra desventaja que aparece en la mayor parte de los esquemas anteriores, reside en la necesidad de conocer, a priori, la ruta necesaria para encaminar las notificaciones a transmitir (de forma similar al servicio ofrecido por un gestor de mensajería). Por este motivo,

si el número de destinos crece, la gestión del servicio de comunicación de alertas será extremadamente compleja, presentando, nuevamente, una escalabilidad mínima. Otros inconvenientes aparecen en aquellos diseños que basan su servicio de intercambio de información en un sistema de inundación, los cuales facilitan el mantenimiento de gestión, pero complican el rendimiento y escalabilidad, pues la complejidad en el intercambio de mensajes crecerá de forma proporcional al número de encaminadores intermedios en el sistema.

La mayoría de estas limitaciones pueden ser solucionadas eficientemente mediante el uso de una infraestructura basada en un modelo de comunicaciones publicador/suscriptor. La mayor ventaja de utilizar este modelo, en comparación a los paradigmas de comunicación anteriormente presentados, radica en la separación entre productores y consumidores de información a intercambiar, y la separación también de toda la infraestructura que implementa el intercambio de mensajes.

Así, podemos evitar los problemas directamente relacionados con la escalabilidad y gestión de la infraestructura de comunicaciones, por medio de la diferenciación de roles, es decir, utilización de publicadores, encaminadores y suscriptores. Un publicador, dentro de dicho esquema, no tendrá que tener ningún conocimiento sobre las entidades que acabarán por consumir la información que va a divulgar sobre el sistema.

De igual modo, los suscriptores de tal información no necesitarán tener ningún conocimiento a cerca de la ubicación física de los productores de información. De esta manera, la inclusión de nuevos servicios en la plataforma de comunicaciones podrá ser efectuada sin que influya en el rendimiento del sistema ni en la interrupción o modificación del servicio o clientes ya existentes.

3.3. Distribución de alertas

Existen diferentes métodos que podemos adoptar a la hora de publicar las alertas en la infraestructura de comunicaciones *publicador/suscriptor*. Dependerá de las necesidades concretas de cada caso elegir un método u otro, y de la tecnología utilizada por los elementos que conforman el Sistema de Detección de Intrusiones.

Cuando un analizador genera una alerta que deberá ser publicada en la infraestructura publicador/suscriptor, sería interesante que la información referente a la alerta publicada no se perdiera una vez los clientes interesados en dicha alerta la hayan recibido. Podrían existir etapas de post-proceso de alertas en las que se requiera de información de alertas anteriores para la evaluación de nuevas alertas, incluso para la utilización de técnicas de aprendizaje para que el Sistema de Detección de Intrusiones sea capaz de adquirir

experiencia en base a las alertas recibidas. Por ello, en nuestra plataforma almacenaremos las alertas detectadas en una base de datos. Esto permitirá un estudio más exhaustivo de las alertas generadas, así como al generación de históricos, estadísticas, etc., que permitirán redefinir (en caso de que sea necesario) las políticas de seguridad del sistema y adecuarlas para ser más tolerantes a los ataques recibidos.

Partiendo de las alertas contenidas en la base de datos (que serán almacenadas en ella por los *Managers* del IDS en el momento oportuno), se nos presentan varias formas de publicar la información que contienen.

3.3.1. Distribución síncrona

La publicación síncrona de mensajes es aquella que se realiza por orden expresa de un cliente publicador o suscriptor; es el propio cliente quien decide qué tipo de alertas y cuando quiere publicarlas o recibirlas. Podríamos adoptar este esquema de distribución síncrona para leer las alertas contenidas en la base de datos en un instante determinado, y publicarlas en la red de comunicaciones, o para obtener las alertas de la base de datos para su estudio y análisis, sin necesidad de publicarlas en la infraestructura de comunicación.

Un esquema de publicación síncrona de alertas podríamos implementarlo mediante *feeds RSS* (*Really Simple Syndication*).

RSS: Reaaly Simple Syndication

RSS es un sublenguaje surgido de la aplicación del metalenguaje XML. RSS corresponde a Rich Site Summary o Really Simple Syndication, y está diseñado para la distribución de noticias o información tipo noticias contenidas en sitios web y weblogs.

Los archivos RSS comúnmente se llaman *feeds RSS* o *canales RSS* y contienen un resumen de lo publicado en el sitio web de origen. Se estructura en uno o más items que constan de un título, un resumen de texto y un enlace a la fuente original en la web donde se encuentra el texto completo. Además, puede incluir información adicional como el nombre del autor o la fecha y la hora de publicación del contenido. Por tanto, cualquier fuente de información susceptible de poder ser troceada en items (los mensajes de un foro, por ejemplo) pueden distribuirse utilizando RSS.

Leyendo el archivo RSS de un sitio web es posible saber si se ha actualizado y con qué noticias o textos, pero sin necesidad de acceder a sus páginas web. El archivo RSS contiene además un enlace específico para cada ítem contenido en el feed que dirige a la página web con el texto completo de la noticia. Para leer los feeds o canales RSS es necesario utilizar un programa llamado *agregador*. Este tipo de programas también se conocen

como lectores de feeds o canales, o agregadores de noticias, entre otras variaciones.

La aplicación habitual del RSS es la distribución de los contenidos o de las noticias de una página web. Esto permite, por ejemplo, incluir los titulares de una página web en otra página web distinta, de modo que es posible y relativamente fácil construir una página web cuyo contenido esté formado por los titulares y resúmenes de los contenidos de una o más páginas web distintas.

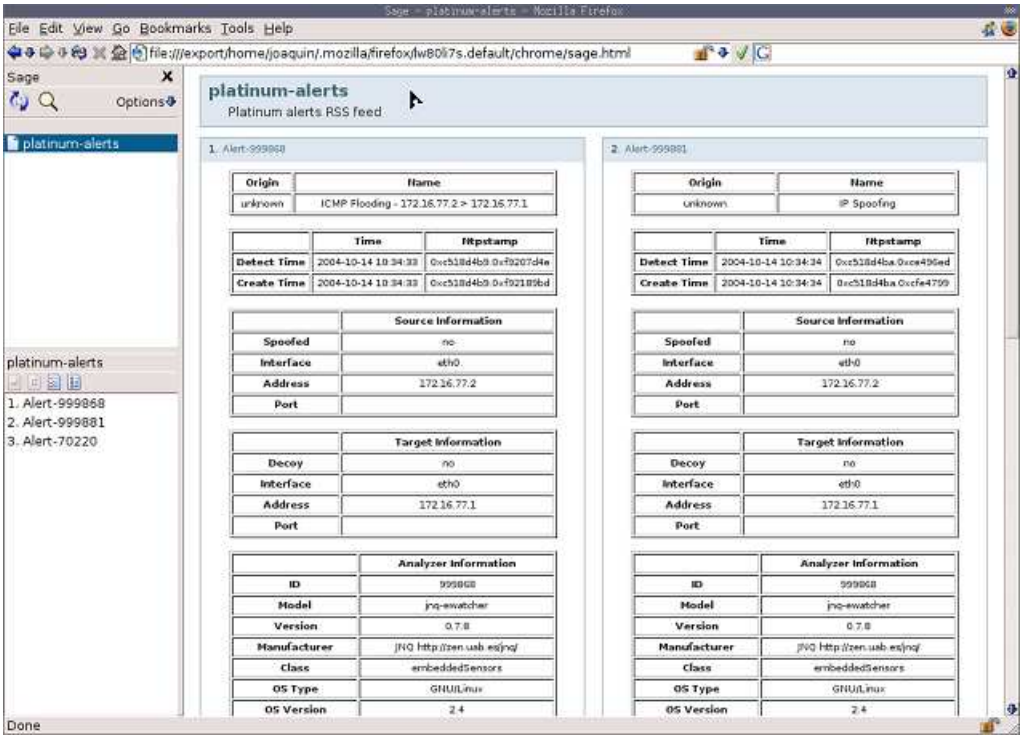


Figura 3.1: Sindicación de alertas

En nuestro entorno de trabajo, como se muestra en la figura 3.1, sería posible configurar diferentes canales RSS que contuvieran la información relacionada con subconjuntos de las alertas generadas, y que terceros clientes de dominios distintos puedan suscribirse a estos canales para descargar las últimas alertas generadas. Además, la adopción de este formato nos permitirá una mejor visualización de las alertas, y permitirá que puedan ser descargadas para su consulta desde otros dominios únicamente con un navegador web o con un agregador de feeds RSS.

3.3.2. Distribución asíncrona

La distribución asíncrona de alertas es aquella en al que los clientes no establecen explícitamente cuando desean publicar/recibir las alertas generadas. A medida que los Analizadores del sistema de detección vayan generando alertas, los Managers se encargarán,

además de almacenarlas en la base datos para que estén disponibles para usos alternativos, de publicarlas en una infraestructura publicador/suscriptor. Así las alertas son publicadas de manera asíncrona en el mismo momento de su creación, y los clientes que estén suscritos a temas o filtros que se correspondan con las alertas publicadas, las recibirán tan pronto como el encaminador local las reciba y pueda entregarlas a los suscriptores.

3.4. Fases de desarrollo

El desarrollo del sistema de comunicaciones de la plataforma se ha hecho de una forma progresiva, considerando las siguientes etapas:

- **Una primera etapa** en la que se desarrollará un sistema de distribución de alertas centralizado, mediante un modelo publicador/suscriptor de tipo *polling pull* basado en distribución por canales o temas, que supone el desarrollo de una primera versión de la infraestructura de comunicación de alertas a utilizar, y el asentamiento de las ideas del proyecto.

- **Una segunda etapa** en la que se desarrollará un segundo cliente a modo de agregador de feeds RSS, que en lugar de comunicar las alertas directamente al Manager, las publique en un encaminador de la infraestructura de comunicaciones publicador/suscriptor para que, por otro lado, un tercer cliente suscrito a alertas mediante filtros sobre el contenido las reciba en forma de notificaciones asíncronas. Este tercer cliente podría ser a su vez un sensor del IDS, que notifique la alerta a un tercer Manager de otro dominio.

- Por último, **una tercera fase** donde un cuarto cliente, a través de *triggers* de la base de datos, acceda a la información en el momento que es almacenada y la publique en la infraestructura de comunicaciones como hacía el cliente de la segunda etapa. Este último punto (como veremos en el apartado de Implementación) resultó ser demasiado complejo y se terminó por desestimar para la realización de este proyecto, ya que no es significativo a la hora de evaluar el sistema de comunicaciones publicador/suscriptor.

De esta forma, tenemos diferentes escenarios a evaluar combinando un enfoque de distribución de alertas centralizado, mediante feeds RSS, y distribuido, mediante publicaciones/subscripciones de tipo push asíncrono en los brokers basado en encaminamiento por contenido.

3.5. Resumen

En este capítulo hemos expuesto los diferentes modelos de comunicación que se podrían adoptar como base para la comunicación entre los componentes de un IDS y hemos expresado los motivos que nos han llevado a decantarnos por el modelo publicador/-suscriptor. Asimismo, hemos presentado las diferentes formas en que las alertas pueden ser publicadas en este sistema, y hemos descrito las fases de desarrollo que hemos seguido para la realización del proyecto.

Capítulo 4

Herramientas utilizadas

Para la implementación de la plataforma, hemos utilizado diversas herramientas y aplicaciones de código abierto que nos proporcionaban la funcionalidad necesaria para poder evaluar el modelo de comunicaciones propuesto. A continuación presentamos los detalles más importantes de cada una de las aplicaciones utilizadas.

4.1. Prelude IDS

Para la gestión de las alertas se ha utilizado Prelude IDS [5]. Prelude es un IDS híbrido (combina sensores basados en red y basados en equipo) que utilizaremos para generar alertas relacionadas con los eventos que lleguen del exterior y almacenarlas en una base de datos. Los elementos que nos proporciona Prelude, son una serie de sensores y managers que configuraremos para que almacenen las alertas en una base de datos de alertas IDMEF.

Los sensores que nos proporciona Prelude por defecto, son el *Prelude-LML*, que es un sensor basado en equipo que monitoriza los ficheros de log en busca de posibles indicios de ataque; y *Prelude-NIDS*, que es un sensor de red, pero que es un poco limitado en comparación con otros sensores de otros IDS. Por ello hemos instalado y configurado Snort [6] para que envíe la información relacionada con los eventos detectados a los managers de Prelude para que éste las almacene en la base de datos.

Los managers de Prelude, podrían ser configurados para que reportaran la información a otros managers de dominios diferentes, con el fin de difundir las alertas detectadas, pero esta característica carece de interés en nuestro diseño, ya que la distribución de alertas se realizará a través del sistema publicador/suscriptor.

Prelude pone también a nuestra disposición una serie de librerías de código abierto que permiten extender su funcionalidad desarrollando nuevos sensores, extendiendo los managers, etc.

Entre estas librerías, utilizaremos la librería *libpreludedb* para almacenar las alertas

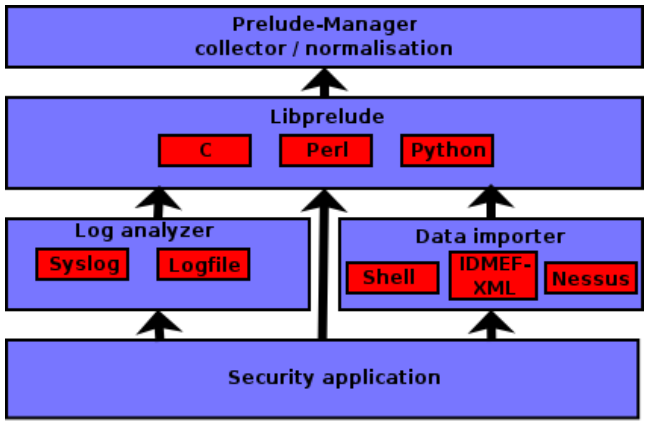


Figura 4.1: Framework de Prelude

en una base de datos IDMEF. La base de datos que hemos elegido para el proyecto es PostgrSQL por ser una base de datos potente, robusta y de código abierto.

4.2. xmlBlaster

Para implementar el subsistema de comunicación publicador/suscriptor hemos utilizado xmlBlaster [7]. xmlBlaster es un MOM (*Message Oriented Middleware*) de código abierto que nos permite implementar una infraestructura de comunicaciones siguiendo el paradigma publicador/suscriptor de manera eficiente. Las principales características que presenta xmlBlaster y que hacen que sea una opción muy interesante a estudiar son las siguientes:

- Es de código abierto, y está licenciado bajo la LGLP (*GNU Lesser General Public License*), lo que permite utilizarlo de manera libre para uso privado, comercial y educacional.
- Soporta diferentes paradigmas de comunicación orientada la paso de mensajes como el publicador/suscriptor o el Punto a Punto.
- El servidor de xmlBlaster (que en nuestra plataforma actuará de encaminador del sistema publicador/suscriptor) está escrito en Java, por lo que es multiplataforma.
- Soporta múltiples protocolos para la comunicación entre los diferentes elementos. Entre ellos destacan: CORBA, RMI, XmlRpc, Raw sockets y e-mail.
- Permite desarrollar clientes en diversos lenguajes de programación tales como C++/C, Java, Python, PHP, Javascript, Perl, C#, Visual Basic.net, entre otros.

- Permite un encolamiento persistente de mensajes en la parte del cliente mediante las APIs de C/C++/Java/ActiveX/Javascript

- Es extensible, ya que la parte de los mensajes xmlBlaster (explicados más adelante) que controla el comportamiento del mensaje está definida en XML.

- Permite clasificar los mensajes en temas, y aplicar filtros XPath sobre ellos para efectuar suscripciones a canales concretos.

- Búsqueda en el contenido de los mensajes basada en el tipi MIME de los mismos. Actualmente hay disponibles plugins para el parseo de los mensajes mediante XPath y expresiones regulares.

- Seguridad independiente del sistema. Soporta autenticación y autorización.

- Existe una arquitectura de plugins para asegurar la persistencia de los datos. Actualmente hay plugins para Oracle, MS-SQLServer, Postgres, Firebird y otros.

4.2.1. Ciclo de vida de los mensajes

Cuando un cliente publicador envía un mensaje a xmlBlaster, dicho mensaje atraviesa una serie de plugins hasta que es almacenado de manera segura en el servidor para su posterior divulgación a los clientes suscriptores, o es rechazado por algún plugin, por ejemplo el de autenticación. Asimismo, cuando un cliente suscriptor se suscribe a un tema, el mensaje es enviado a través de otra serie de plugins antes de llegar al destinatario.

En la figura 4.2 se muestra un esquema del camino que siguen los mensajes. Los plugins que aparecen en color gris, son necesarios: si no se registran explícitamente se utilizarán unos por defecto. Los plugins que aparecen en color verde son opcionales y registrados en el arranque de xmlBlaster. Por último, los plugins de color marrón permiten controlar el comportamiento de los clusters o grupos de encaminadores que comparten información. Son obligatorios si se utiliza este tipo de arquitectura.

Los plugins que atraviesa el mensaje, pueden dividirse en cuatro etapas:

Recibimiento del mensaje por parte del encaminador

El primer plugin por el que pasan los mensajes, es el plugin referente al **protocolo** utilizado para la comunicación (CORBA, RMI, Socket, etc.). El protocolo utilizado para conectar y enviar los mensajes a un encaminador xmlBlaster y el utilizado por éste para

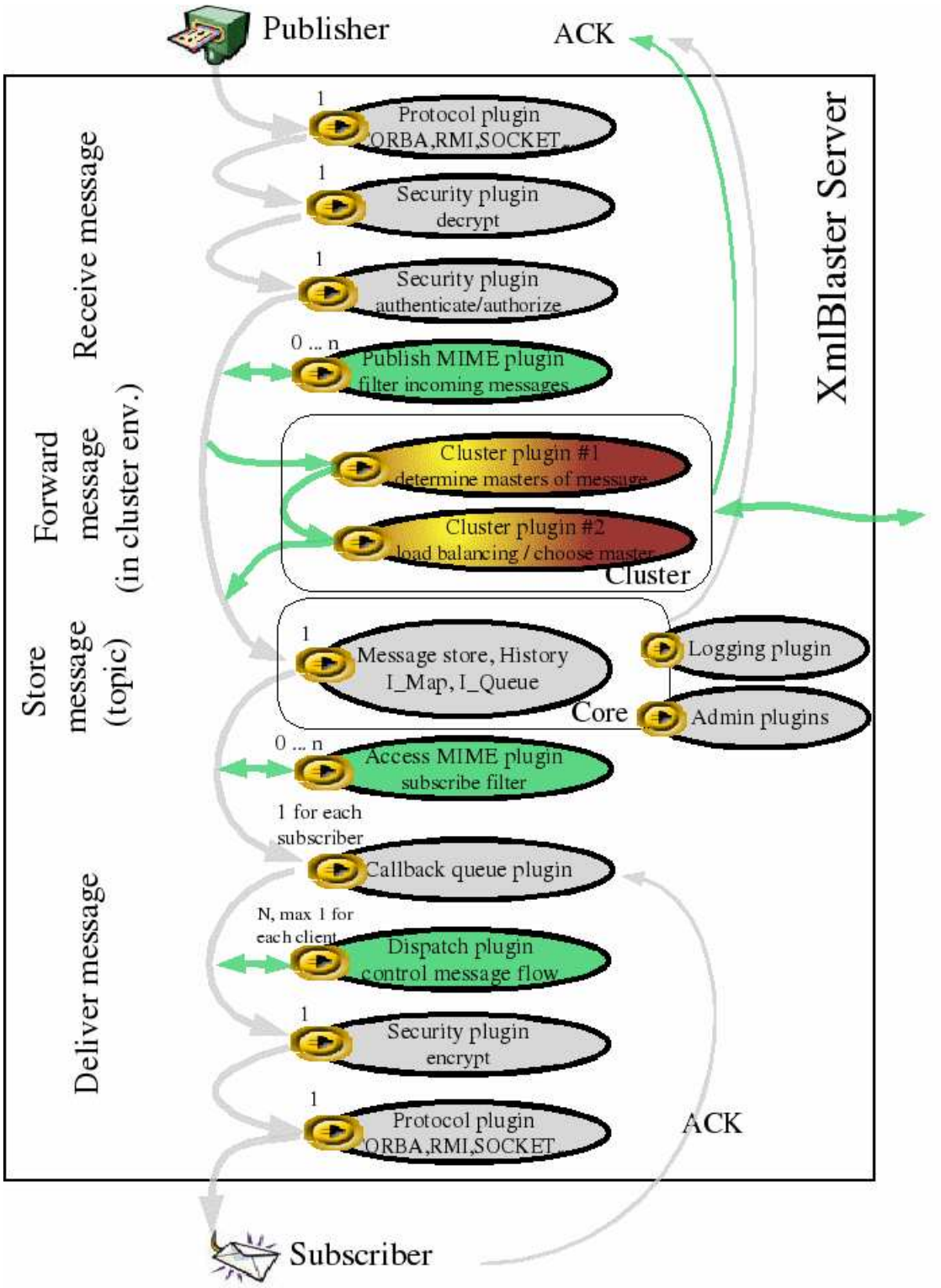


Figura 4.2: Ciclo de vida de los mensajes

enviar los mensajes a los clientes suscriptores, no tiene por qué ser el mismo. Por ejemplo, podríamos conectar a un encaminador xmlBlaster y mandar un mensaje utilizando el protocolo RMI, y éste entregar el mensaje utilizando el e-mail o HTTP.

El siguiente plugin que atraviesan los mensajes que llegan al encaminador es el plugin de **seguridad**. Este nace de la necesidad de mantener la *confidencialidad* e *integridad* [18] de los datos enviados. Ambas son los objetivos principales del plugin de seguridad, aunque para lograrlos han de considerarse otros más básicos como: responsabilidad, no-repudio, privacidad y anonimato.

Una vez pasado el plugin de seguridad, se envía al **MIME plugin**. Éste sirve para filtrar mensajes y modificarlos antes de que xmlBlaster lo publique en la red de encaminadores.

Reenvío del mensaje (en entornos de cluster)

Tras haber sufrido las modificaciones necesarias, el mensaje ya está preparado para ser distribuido por la red de encaminadores. De esto se encarga el plugin de **cluster**, en caso de que lo tengamos configurado. Podemos ver que este paso consta de dos etapas: la primera consiste en una búsqueda de los encaminadores *master* de la red de encaminadores; y la segunda consiste en realizar el balanceo de carga y elegir el nodo *master* al que será entregado el mensaje. Es evidente que aquí influirá el diseño que se haya hecho de la red de encaminadores y de las reglas de encaminamiento que se hayan definido.

Almacenamiento del mensaje y tema

El siguiente paso es almacenar el tema en xmlBlaster. Si el tema aún no existía, se creará una nueva entrada para él; si ya existía, se asignará el nuevo mensaje al tema ya existente. En la figura 4.3 puede verse el ciclo de vida de un tema en xmlBlaster desde su creación hasta su eliminación.

No hay que confundirlo con el ciclo de vida de los mensajes. Un tópico podríamos entenderlo como un parámetro bajo el que se clasificarán los diferentes mensajes que lleguen al encaminador. En la figura 4.4 podemos ver el ciclo de vida de un mensaje que ha sido publicado bajo un tema en concreto.

El plugin que se encarga del almacenamiento de los mensajes, interactúa a su vez con los plugins de administración y *logging*. El primero se encarga de realizar tareas de administración como monitorizar los mensajes, el rendimiento, el tamaño de las colas, etc. El segundo se encarga de registrar las acciones que ocurren en el encaminador xmlBlaster.

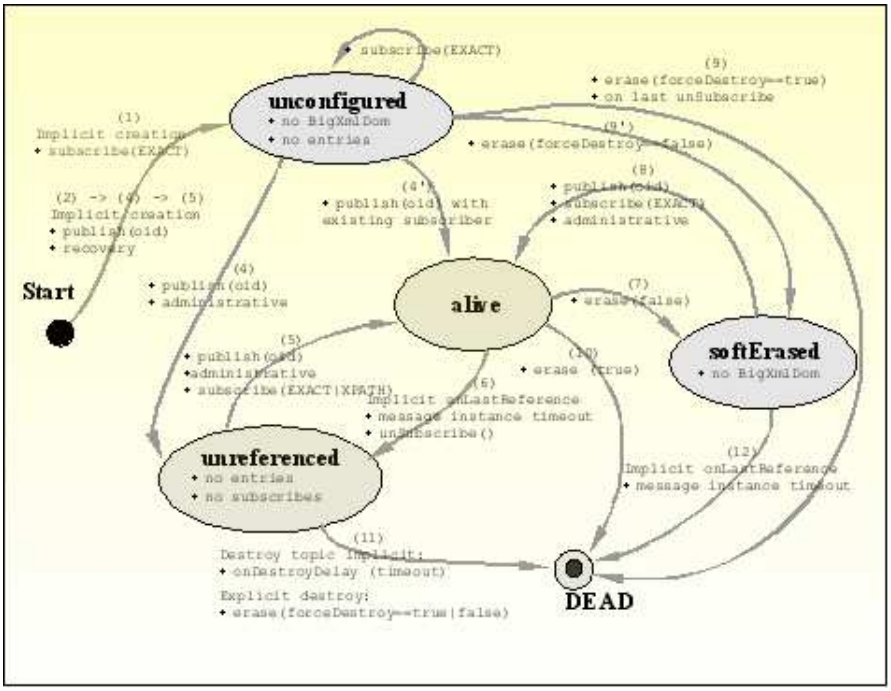


Figura 4.3: Ciclo de vida de los temas de los mensajes

Entrega del mensaje

El primer plugin que atraviesa un mensaje en esta etapa es el plugin de **acceso MIME**. Es aquí donde se realiza la búsqueda en el contenido del mensaje mediante expresiones regulares o filtros XPath. Los filtros se configuran en el campo QoS del mensaje (explicado más adelante).

El siguiente plugins que atraviesa es el plugin de **encolamiento**. Este plugin sirve para almacenar en una cola de manera temporal el mensaje hasta que es entregado al destinatario, o para almacenarlo de forma persistente de manera que pueda recuperarse tras una caída del encaminador. Las colas existen tanto en el lado del servidor como en el lado del cliente. Así si un cliente publicador pierde la conexión con el encaminador, podrá encolar los mensajes para que sean entregados cuando la conexión se recupere.

Tras ser encolado, el mensaje es enviado al plugin de **control de flujo**, que se encargará de enviar los mensajes a los diferentes clientes de la forma adecuada.

Por último, los mensajes pasarán por el plugin de seguridad para ser codificados en caso de que sea necesario, y finalmente se entregarán al plugin referente al **protocolo**, para enviar el mensaje al destinatario utilizando el protocolo adecuado.

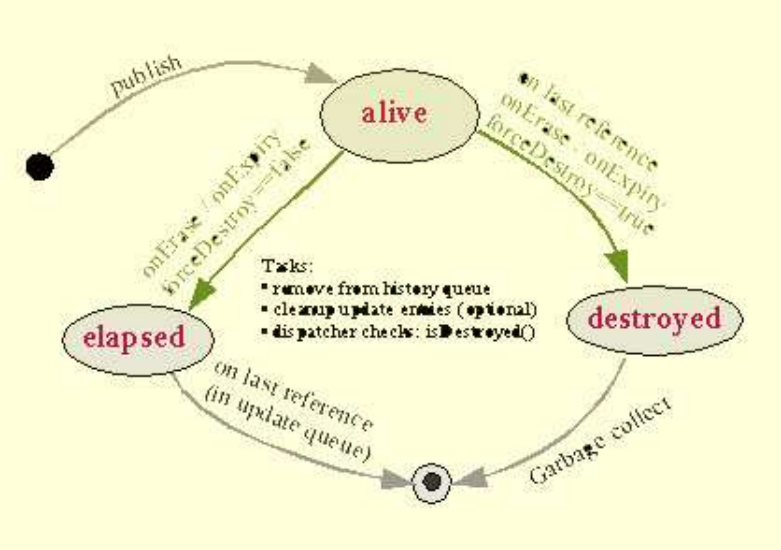


Figura 4.4: Ciclo de vida de los mensajes de un tema concreto

4.2.2. Estructura de los mensajes

xmlBlaster utiliza el formato XML para el intercambio de mensajes, pero no se limita solamente al envío de texto plano; es capaz de enviar todo tipo de ficheros, imágenes, etc. Depende de los clientes publicadores y suscriptores el tipo de datos y la información que se intercambia encapsulada en estos mensajes. Asimismo, los mensajes de xmlBlaster pueden tener diferentes significados: aparte de los mensajes de publicación y suscripción por parte de los clientes, existen también mensajes de control, como por ejemplo mensajes de autenticación, de *keep-alive*, etc. Todos estos mensajes siguen el mismo ciclo de vida explicado en el mensaje anterior y comparten el mismo formato.

Cada mensaje de xmlBlaster se compone de tres partes esenciales:

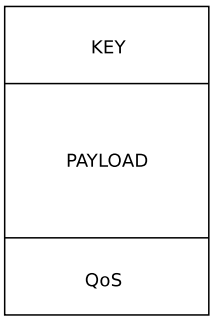


Figura 4.5: Formato de los mensajes xmlBlaster

El campo key

Sirve para indicar a xmlBlaster los criterios bajo los que este mensaje será identificado, así como el tipo MIME del payload. Las propiedades de este campo se especifican en formato XML y son las siguientes:

- *oid*: Es el identificador o tema del mensaje. Los mensajes publicados en un broker xmlBlaster son clasificados en temas. Cuando llega un nuevo mensaje, el broker comprueba si el tema (oid) del mensaje publicado ya existe; si no existe, crea un nuevo tema bajo el que clasificar los mensajes que lleguen con ese mismo oid. Si ya existiera, almacenaría el mensaje bajo el tema ya existente en una estructura de árbol.

- *contentMime*: Especifica el tipo MIME del payload del mensaje.

- *queryType*: Si el mensaje que se envía al broker es una suscripción, con este campo se indica de qué modo quiere realizarse. Admite diferentes valores que veremos más adelante, cuando se explique en qué consisten las suscripciones.

Aparte de estos atributos del elemento key, podemos añadirle elementos XML para hacer este campo más específico. De este modo, no sólo dispondríamos del tema para clasificar los mensajes, sino que además tendremos una serie de tags XML con los que discriminar los mensajes.

```
<key oid="high" contentType="text/xml">
   <Classification>SNMP trap tcp</Classification>
</key>
```

Ejemplo del campo KEY

El campo payload

En él se publicará el contenido del mensaje que queremos enviar, de acuerdo al tipo MIME especificado en el campo key. Por defecto se considera que el tipo MIME de los mensajes enviados es texto plano (*text/plain*). En nuestro sistema, utilizaremos el campo *payload* para publicar en la infraestructura publicador/suscriptor las alertas en formato IDMEF generadas por los analizadores de Prelude.

El campo QoS

Sirve para indicarle a xmlBlaster cómo debe administrar el mensaje. En él especificamos opciones como la duración del mensaje en el broker, la prioridad, el destinatario del mensaje en el caso de que sea una publicación Punto a Punto, el publicador del mensaje, si se desea publicar únicamente partes concretas del mensaje, los filtros y plugins que serán aplicados a dicho mensaje, etc.

Para nuestra aplicación, la parte más interesante de este campo es la especificación de los plugins que van a ser utilizados para tratar el mensaje. La definición y datos que pueden transportarse en el campo QoS es un tanto compleja; a continuación, mostraremos con unos ejemplos los elementos más interesantes que puede contener dicho campo dependiendo del tipo de mensaje.

Ejemplos del campo QoS

En el siguiente ejemplo de campo QoS podemos ver un ejemplo de especificación de las propuedades globales de los campos como el tiempo de vida del mensaje en el servidor.

```
<qos>
  <!-- Borrar el mensaje al entregarlo -->
  <isVolatile/>

  <!-- Restaurar el mensaje tras un reinicio -->
  <isDurable/>

  <!-- Tiempo de expiración del mensaje -->
  <expiration remainingLife='990'/>
</qos>
```

A la hora de publicar mensajes, es posible establecer la prioridad y autor del mensaje utilizando los elementos *<sender>* y *<priority>* como en el ejemplo siguiente. La prioridad es un entero que va desde el 0 al 9, siendo 0 la prioridad más baja, y 9 la prioridad más alta.

```
<qos>
  <sender>admin</sender>
  <priority>5</priority>
</qos>
```

Si es un mensaje Punto a Punto, se deberá especificar el destinatario del mismo. Si el usuario de destino no está identificado en el sistema, el mensaje se descartará pasado un cierto intervalo de tiempo. Una posible configuración de mensaje dirigido sería la siguiente, en la que se especifica el tiempo de expiración, en milisegundos:

```
<qos>
  <destination queryType='EXACT'>
    Administrador
    <ForceQueuing timeout='1200'/>
  </destination>
</qos>
```

Para suscribirnos a un mensaje basándonos en una consulta XPath sobre el campo *key*, se utilizaría también el campo destination, pero especificando que el tipo de consulta que se va a realizar sobre el mensaje será una consulta XPath. Por ejemplo:

```
<qos>
  <destination queryType='XPath'>
    //role[@id='Administrador']
  </destination>
</qos>
```

También podemos limitar el tamaño de la cola de mensajes de llegada y establecer así el número máximo de mensajes que queremos recibir. Es posible también limitar el tamaño de los mensajes a recibir. Una posible configuración para establecer el máximo número de mensajes a 1000, o exigir que no se sobrepasen los 4000Kb de *payload* seria:

```
<qos>
  <queue relating='unrelated'
         maxMsg='1000'
      maxSize='4000'
      onOverflow='deadLetter' />
</qos>
```

Por último, veremos un ejemplo de aplicación de filtros sobre el contenido del mensaje. En el ejemplo siguiente se utiliza un filtro XPath para seleccionar únicamente aquellos mensajes que en el contenido tengan un elemento *Impact* cuyo atributo *severity* valga *high*.

```
<qos>
  <filter type='XPathFilter'>
    //Impact[@severity='high']
  </filter>
</qos>
```

4.2.3. Plugins utilizados

Como hemos visto hasta ahora, el comportamiento de xmlBlaster viene definido por una serie de plugins que los mensajes atravesarán antes de llegar al destino. A continuación daremos una visión más específica de aquellos plugins que tienen un especial interés en nuestra aplicación.

MIME access XPath

xmlBlaster nos permite acceder al contenido de los mensajes y filtrarlos mediante expresiones XPath por medio del plugin *MIME access XPath*. Los filtros pueden efectuarse sobre el campo *payload* o sobre el campo *QoS* del mensaje. Por defecto se filtrará sobre el campo *payload*, pero podemos indicar lo contrario con la opción *matchAgainstQos=true*. En el caso de filtros sobre el campo *payload* sólo serán válidos si el tipo MIME del campo es *text/xml*.

Este filtro está disponible tanto para peticiones síncronas como asíncronas (las suscripciones típicas de un modelo publicador/suscriptor). En ambos casos, el filtro debe especificarse en el campo QoS del mensaje de suscripción enviado.

Si se desean hacer consultas más complejas, pueden encadenarse una serie de filtros en el campo QoS, que serán ejecutados secuencialmente en forma de AND lógica; solamente se devolverá el mensaje si se cumplen todos los filtros especificados.

Por último, hay que remarcar que todos los mensajes de suscripción llevan, además del campo QoS, los campo Key y Payload correspondientes. En el campo Key estarán especificados un tema o una consulta XPath para efectuar sobre el tema de los mensajes. Este filtrado se realiza primero, y se registra el cliente como suscritor de los mensajes de los temas correspondientes a los criterios especificados. Una vez hecho esto, se aplicarán sobre este subconjunto los filtros de contenido definidos en el campo QoS.

Para habilitar este plugin es necesario especificar, como mínimo, en el fichero de configuración de xmlBlaster la directiva:

MimeAccessPlugin[XPathFilter][1.0]=org.xmlBlaster.engine.mime.xpath.XPathFilter

4.2.4. Clustering

Una de las principales características que proporciona xmlBlaster es la capacidad de formar *clusters* o grupos de encaminadores que se comunican e intercambian mensajes entre ellos. Cada encaminador conoce a sus encaminadores vecinos y intercambia mensajes con ellos, que a su vez pueden comunicarse con terceros encaminadores. La manera en que xmlBlaster implementa este modelo de comunicaciones es mediante un sistema de encaminadores *maestro/esclavo*.

Este sistema permite distribuir de manera muy flexible los encaminadores por nuestra red, y definir reglas de encaminamiento de una manera sencilla. El sistema maestro/esclavo implementado por xmlBlaster nos ofrece las siguientes características:

- **Escalabilidad**: Un encaminador maestro puede tener múltiples encaminadores esclavos, y estos esclavos pueden tener a su vez otros encaminadores esclavos (un encaminador esclavo puede ser a su vez el maestro de otros). Esto permite distribuir los mensajes prácticamente a un número ilimitado de clientes.

- **Separación lógica entre dominios**: Una instancia de encaminador xmlBlaster puede ser maestro únicamente determinados subconjuntos de mensajes, y esclavo de otros subconjuntos. Los subconjuntos de mensajes de los que un encaminador es maestro o esclavo puede determinarse por el dominio del mensaje, por el tema del mensaje, e incluso mediante filtros XPath. Así, un mensaje puede ser distribuido únicamente a un conjunto determinado de encaminadores, liberando la carga del resto.

Otras características relacionadas con los entornos de cluster que son consideradas en xmlBlaster, pero que aún no están disponibles en las versiones actuales son:

- **Tolerancia a fallos**: Cuando un encaminador deja de ser accesible por los motivos que sean, otros encaminadores deberían poder asumir su papel de manera transparente, y notificar a los clientes que estuvieran conectados de forma local a dicho encaminador la nueva situación.

- **Balanceo de carga**: Una de las situaciones más comunes es tener encaminadores de un mismo nivel que realizan balanceo de carga entre ellos. Los mensajes publicados serán destinados a un encaminador u otro según diferentes algoritmos de balanceo.

Estas dos últimas características serían muy interesantes para nuestra plataforma. La primera porque permitiría la reconfiguración automática del encaminamiento entre los

elementos de la plataforma ante un posible fallo, y la segunda porque supondría un sistema completamente distribuido. Con las versiones actuales de xmlBlaster y el modelo de comunicaciones maestro/esclavo pueden definirse reglas de encaminamiento basadas en dominios de manera sencilla y crear grandes redes de encaminadores, pero sigue existiendo el problema de la jerarquía de encaminadores. De todos modos, ambas características estarán presumiblemente disponibles en futuras versiones, así que por el momento utilizaremos las características de clustering del sistema actual, a la espera de una actualización que en principio no debería ser muy costosa.

Características de los clusters

Una parte importante en los entornos de clustering, es el descubrimiento y búsqueda de otros encaminadores que pertenecen al cluster: cómo encontrarlos otros encaminadores y cómo acceder a la información que contienen. Esto lo realiza internamente xmlBlaster utilizando el modelo publicador/suscriptor. Cada nodo del cluster guarda su información en mensajes xmlBlaster (con sus campos key, payload y QoS), y los encaminadores pueden suscribirse a ellos como si fueran clientes normales de una red de comunicaciones publicador/suscriptor.

A continuación se presenta el comportamiento de los clusters de xmlBlaster y las principales características y funcionalidades que ofrecen:

- **Múltiples encaminadores**: Si existen numerosas instancias de xmlBlaster, cada mensaje es asignado únicamente a un encaminador maestro (si no está activado el balanceo de carga, por el momento no disponible). El resto de encaminadores se conectan y autentican entre ellos como un cliente más y actúan como esclavos para aquellos mensajes de los que no son maestros.

- **Conexión entre encaminadores**: La conexión entre encaminadores se realiza de forma pasiva. Tan pronto como un cliente se suscribe a un subconjunto de mensajes sobre los que el encaminador no es el maestro, éste se conecta al encaminador maestro y obtiene los mensajes correspondientes a la suscripción. En adelante, las peticiones relacionadas con estos mensajes se resolverán de forma local.

- **Encaminamiento de mensajes publicados**: Si un mensaje es publicado por un cliente a un encaminador esclavo, éste es reenviado a su encaminador maestro (que a su vez puede ser encaminador esclavo y reenviar el mensaje a un tercer encaminador, y así sucesivamente). Si el encaminador maestro no está disponible, el mensaje se encola en el encaminador esclavo hasta que pueda ser entregado.

- **Descubrimiento del maestro basado en plugins**: xmlBlaster resuelve el problema del descubrimiento del encaminador maestro mediante los plugins presentados en apartados anteriores: pueden hacerse consultas sobre el tema de los mensajes, consultas sobre el dominio, y consultas mediante expresiones XPath. Más adelante veremos varios ejemplos de configuraciones de entornos de cluster en los que se utilizarán este tipo de filtros.

- **Interfaz para el balanceo de carga**: xmlBlaster proporciona una interfaz para el balanceo de carga. Por defecto el algoritmo que se utilizará es un *Round Robin*, pero pueden definirse algoritmos propios para realizar el balanceo. Esta característica también pertenece a futuras versiones de xmlBlaster.

- **Maestro por defecto**: Un encaminador xmlBlaster es por defecto el maestro para todos los mensajes publicados por los clientes locales, a menos que en dichos mensajes se especifique un dominio diferente, o se haya definido expresamente otro encaminador como maestro de ese conjunto de mensajes. El comportamiento como encaminador maestro por defecto puede habilitarse y deshabilitarse desde la configuración inicial o mediante mensajes xmlBlaster.

- **Maestro desconocido**: Si no se encuentra un encaminador maestro para un mensaje, es encolado y almacenado de forma local. Cuando aparezca un encaminador maestro para el mensaje, será desencolado y enviado.

- **Encaminamiento punto a punto**: Es posible indicarle a xmlBlaster que realice una entrega directa de un mensaje, especificando el nombre del nodo del cluster al que desea ser enviado. Cuando esto se produce, se buscarán todos los encaminadores vecinos y si el deseado se encuentra entre ellos, se le enviará el mensaje; si no es encontrado, el mensaje será encaminado de la forma habitual.

Configuración de entornos de cluster

La configuración de los entornos de cluster puede realizarse en el fichero de configuración de xmlBlaster o mediante el intercambio de mensajes. La forma más cómoda y entendedora de hacerlo es configurando las propiedades de cada nodo del cluster en su fichero de configuración. A continuación se introducirán mediante unos ejemplos las configuraciones de cluster más comunes, y se explicará la sintaxis y el significado de las diferentes etiquetas utilizadas.

```
cluster.node.id=dokken
```

```
cluster.node.master[dokken]=\
   <clusternode id='dokken'>\
      <master type='DomainToMaster' acceptOtherDefault='true'>\
         <key queryType='DOMAIN' domain='Alert'/>\
         <key queryType='DOMAIN' domain='Heartbeat'/>\
      </master>\
   </clusternode>
```

En este ejemplo se define un nodo del cluster cuyo ID es *dokken*. Usualmente este ID puede ser la dirección IP o el nombre de dominio del encaminador. Seguidamente se define el conjunto de mensajes para los que este nodo será maestro; los mensajes cuyo dominio sea *Alert* o *Heartbeat* tendrán a este nodo como maestro. Además, con la directiva *acceptOtherDefault='true'* indicamos que aquellos mensajes que no tengan un dominio especificado serán asignados también a este nodo.

El establecimiento del dominio de los mensajes se realiza mediante un nuevo atributo del campo *key* de los mensajes xmlBlaster. Por ejemplo, para publicar un mensaje en el dominio *Alert* se utilizaría un mensaje con un campo *key* como el siguiente:

```
<key oid='Tema del mensaje' domain='Alert'/>
```

En el siguiente ejemplo puede verse cómo se especifica en la configuración de un nodo del cluster el modo en que puede ser encontrado por sus nodos vecinos:

```
cluster.node.id=dokken
cluster.node.info[dokken]=\
   <clusternode id='dokken'>\
      <connect><qos>\
         <address type='IOR' bootstrapPort='7601'/>\
         <address type='SOCKET'>192.168.1.4:7701</address>\
      </qos></connect>\
   </clusternode>
```

En este caso hemos definido dos tipos de direcciones necesarias para acceder al nodo *dokken*: una del tipo SOCKET, que es el protocolo que utilizarán comúnmente los clientes de nuestra plataforma para conectarse al encaminador, y otra dirección de tipo IOR, que es la que utilizan por defecto los encaminadores de xmlBlaster para comunicarse mediante CORBA. En la dirección de tipo IOR se ha omitido el nombre de host, ya que por defecto se utiliza la dirección de la propia máquina.

En el ejemplo que sigue se muestra una configuración más completa del nodo *dokken*, indicándole también la información que necesita para acceder a su nodo vecino *maqui*.

```
cluster.node.id=dokken
```

```
cluster.node[dokken]=\
   <clusternode id='dokken'>\
      <connect><qos>\
         <address type='IOR' bootstrapPort='7601'/>\
         <address type='SOCKET'>192.168.1.4:7701</address>\
      </qos></connect>\
      <master type='DomainToMaster'>\
         <key queryType='DOMAIN' domain='Alert'/>\
         <key queryType='DOMAIN' domain='Heartbeat'/>\
      </master>\
   </clusternode>

cluster.node[maqui]=\
   <clusternode id='maqui'>\
      <connect><qos>\
         <address type='IOR'\
              bootstrapHostname='192.168.1.5'\
          bootstrapPort='7602'/>\
         <address type='SOCKET'>192.168.1.5:7702</address>\
      </qos></connect>\
      <master type='DomainToMaster'>\
         <key queryType='EXACT' oid='Alert'/>\
         <key queryType='XPATH'>//Heartbeat'</key>\
      </master>\
   </clusternode>
```

Este caso es más complejo que el anterior y mucho más habitual. Primero se establece el ID del nodo dentro del cluster a *dokken*, y seguidamente se especifica (en un mismo bloque) tanto la información relativa a las direcciones mediante las cuales será accesible el nodo, como el subconjunto de mensajes para los que el nodo es maestro (en este caso para los mensajes cuyo dominio es *Alert* o *Heartbeat*. Seguidamente se proporciona la información del nodo *maqui*. Se especifican las direcciones necesarias para poder acceder a él, y se configura el subconjunto de mensajes para los que *maqui* es el nodo maestro. El nodo actual (*dokken*) hará de esclavo de este subconjunto de mensajes y se los enviará a *maqui*. Puede verse que el subconjunto de mensajes que ha de enviarse a *maqui* no se ha definido a partir del dominio de los mensajes, sino en consultas basadas en el campo *key*. La sintaxis de estos filtros es la misma que la comentada en apartados anteriores.

La configuración de los nodos de cluster puede ir más allá, especificando criterios más complejos como el orden de los nodos en la jerarquía de maestros/esclavos, los nodos de *backup* que deberán asumir el papel del propio nodo en caso de fallo (esta característica

aún no está implementada en la versión actual), etc. No obstante, no comentaremos estas características, ya que para los objetivos de la plataforma no es necesaria una configuración tan precisa; simplemente se necesita poder crear una estructura de encaminadores adecuada, y definir las reglas de encaminamiento entre ellos.

4.3. Resumen

En este capítulo se ha dado una visión de las herramientas que se ha utilizado para la implementación de la plataforma de comunicaciones. Se han presentado las características de cada una y explicado los conceptos básicos y el funcionamiento de las mismas, y se ha intentado mostrar el potencial y flexibilidad de xmlBlaster como framework sobre el que implementar la red de comunicaciones publicador/suscriptor.

Capítulo 5

Diseño e implementación

En este capítulo describiremos los elementos de la plataforma de gestión de ataques y su interacción. En la siguiente figura se muestra un esquema de la plataforma y los elementos que serán descritos a continuación.

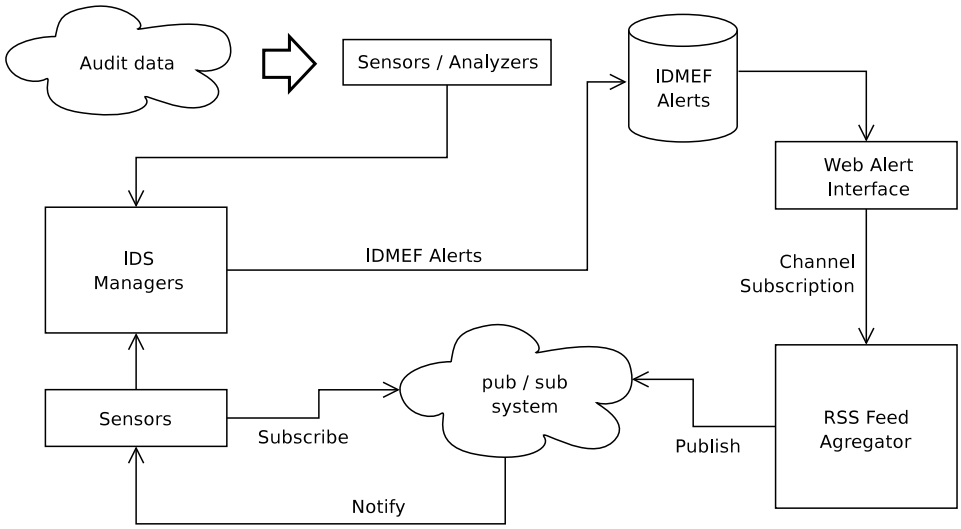


Figura 5.1: Esquema general de la plataforma de detección de ataques

5.1. Elementos principales de la plataforma

Los elementos principales de nuestra plataforma para la gestión de ataques coordinados son los que se encargan de generar las alertas en el formato IDMEF para ser almacenadas en la base de datos, y los que se encargan de recoger la información de las diferentes alertas para correlacionarlas y generar los posibles escenarios de ataque.

5.1.1. Analizadores

Los analizadores (junto con sus respectivos sensores o unidades de detección) son los elementos encargados de notificar alertas locales a partir de un conjunto de información de auditoría local. Su tarea consiste en analizar la información recogida por los sensores de red o de sistema y almacenarla en una base de datos de alertas en formato IDMEF.

5.1.2. Managers

La utilización de múltiples analizadores en un entorno distribuido, con distintas técnicas de detección en sus unidades de detección (sensores), incrementa las probabilidades de detectar un mayor número de indicios de ataque, pero dificulta, a su vez, el proceso de gestión y tratamiento de alertas, tanto locales como externas al entorno donde están instalados los distintos componentes. Por ello, es necesaria la utilización de un conjunto de gestores de alertas, que se encarguen de realizar de la forma adecuada un proceso de fusión, agregación y correlación de alertas.

Durante el proceso de fusión, todas aquellas alertas que apunten a un mismo evento, reportado por distintos analizadores de la plataforma, serán agrupadas y transformadas, en un proceso posterior de agregación, en una única alerta que reportará de forma unitaria el evento. A continuación, la realización del proceso de correlación de alertas permitirá encontrar enlaces lógicos que relacionen distintas actividades de un mismo escenario de ataque, apuntando hacia el objetivo perseguido por dicho ataque, y posibles acciones que podrían detenerlo.

5.2. Modelos de difusión de alertas

Como hemos visto en el apartado de análisis de requerimientos, tenemos diferentes modelos para la difusión de las alertas entre analizadores y managers utilizando el modelo publicador/suscriptor. La difusión de mensajes síncrona basada en canales, y la difusión de mensajes asíncrona basada en contenido, son los dos sistemas de comunicación publicador/suscriptor que conviven en la plataforma de detección. A continuación, se dará una visión global de las características de cada uno.

5.2.1. Publicación y suscripción de alertas locales basada en canales

El conjunto de alertas almacenadas en la base de datos de cada entorno de detección, puede ser visualizado y exportado al resto de componentes a partir de un esquema

de sindicación de alertas basado en RSS (*Really Simple Syndication*) [8]. Los clientes realizarán suscripciones a los diferentes canales RSS y periódicamente irán realizando sondeos para descargarse las novedades. El contenido del feed RSS, generado directamente a partir de las alertas IDMEF de la base de datos local, es codificado y organizado mediante lenguaje XML. De esta forma las alertas IDMEF podrán ser integradas de forma unitaria mediante cualquier agregador de canales RSS.

A través de un agregador de canales RSS diseñado específicamente para nuestra plataforma, las alertas publicadas a través de canales RSS pueden ser de nuevo publicadas, en caso de que el administrador así lo desee, sobre el segundo sistema publicador/suscriptor que introduciremos en el siguiente apartado. Las alertas publicadas sobre este segundo sistema, más completo y eficaz a la hora de realizar búsquedas y suscripciones, podrán ser accedidas por los managers del resto de entornos de la plataforma, haciendo posible las tareas de cooperación necesarias para llevar a cabo el proceso de detección descentralizado. Este agregador de canales RSS específico de cada entorno se encargará de realizar las suscripciones, sondeos, y publicaciones necesarias, según la política de administración configurada por el responsable de la plataforma de gestión.

5.2.2. Publicación y suscripción de alertas globales basada en contenido

La comunicación entre los diferentes managers de la plataforma que realizan las tareas de cooperación y correlación descentralizada de alertas será proporcionada por *xmlBlaster*.

Las alertas serán encapsuladas en mensajes xmlBlaster, descritos en el capítulo de Herramientas, estableciendo una cabecera sobre la que se aplicará un primer filtrado y se utilizará para clasificar el mensaje, un contenido que será la alerta en formato IDMEF, y una sección de control en la que se colocarán los valores para determinar el comportamiento del mensaje, así como los filtros sobre el contenido del mismo, en caso de ser un mensaje de suscripción.

Los filtros para la suscripción de alertas serán formulados en forma de expresiones XPath, que serán evaluadas sobre el contenido de los mensajes, para que cada encaminador pueda decidir sobre qué suscripciones ha de ser entregado el mensaje.

Las posibles interacciones ofrecidas por parte de xmlBlaster a los clientes conectados al sistema son mostradas en la figura 5.2. Para publicar alertas, los clientes de la plataforma invocarán la operación de entrada *pub(a)*, pasando la alerta *a* como parámetro. Una vez publicada, la alerta será entregada por el encaminador del sistema a los suscriptores

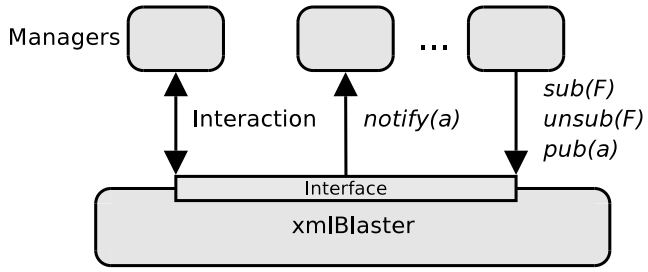


Figura 5.2: Interacción de los managers a través de xmlBlaster

adecuados, a través de la operación de salida *notify(a)*.

Los suscriptores expresarán su interés en recibir alertas específicas, mediante la utilización de la operación *sub(F)*, siendo *F* el filtro XPath pasado como parámetro a la operación. Cada cliente podrá tener múltiples suscripciones activas, las cuales podrán ser revocadas de manera independiente a partir de la operación *unsub()*.

Todas estas operaciones son instantáneas y toman sus argumentos a partir del conjunto de clientes $\mathcal{C}$, del conjunto de alertas $\mathcal{A}$ y del conjunto de filtros $\mathcal{F}$. De manera formal, podemos expresar un filtro $F \in \mathcal{F}$ como la siguiente expresión booleana:

$$\mathcal{F}: \qquad a \longrightarrow \{true, false\} \qquad \forall a \in \mathcal{A}$$

De esta forma, una *notificación N* aplica sobre un *filtro* $F \in \mathcal{F}$ si y sólo si $F(a) = true$. Asumimos que una alerta será publicada de forma única sobre el sistema y que un filtro será asociado a una alerta mediante un identificador único, de manera que el encaminador encargado de la entrega será capaz de identificar tal asociación.

Para nuestra plataforma, los distintos managers de cada entorno podrán registrar su interés en un subconjunto $\mathcal{A}$ de las alertas publicadas sobre la infraestructura de alertas invocando la operación *sub(A)*, que tomará el filtro *A* como parámetro, con

$$A(a) = \begin{cases} true & , \quad a \in \mathcal{A} \\ false & , \quad a \notin \mathcal{A} \end{cases}$$

Una vez suscritos a estos filtros, el sistema entregará a los managers aquellas alertas publicadas a través del agregador de RSS que sean aplicables a estos filtros.

5.3. Configuración e implementación

Tal como veíamos en el apartado de análisis, se ha dividido el proyecto en tres etapas diferentes, para ir implementando la infraestructura de comunicaciones necesaria para el intercambio de alertas. Veremos a continuación las tareas de instalación, configuración

que se realizaron para poner a punto la plataforma así como las aplicaciones y librerías creadas para poder evaluar su funcionamientos.

5.3.1. Preparación del sistema

El sistema operativo elegido para la instalación de la plataforma durante la fase de desarrollo es Debian GNU/Linux (*www.debian.org*). Es una distribución de las más estables que pueden encontrarse y es la que sigue más de cerca los principios del Open Source. Cuenta también con gran cantidad de software en sus repositorios y la mayoría de las librerías que serán necesarias para instalar las aplicaciones que se usarán en el proyecto.

- PostgreSQL 7.4
- xmlBlaster 1.0.7
- Snort 2.4.3
- Prelude IDS 0.9.0-1
  - libprelude0
  - libprelude2
  - libpreludedb0
  - prelude-manager
  - prelude-lml sensor
  - prelude-nids sensor
- libIDMEF-1.0.2-beta1
- libxml2-2.6.16-7

Se ha configurado Snort para que envía la información relacionada con los eventos detectados a Prelude, y se ha registrado en este último un nuevo sensor para que recoja la información recibida de Snort.

Además del software básico, se ha instalado un CVS (*Control Version System*) para seguir el desarrollo de las diferentes partes del proyecto, junto con ViewCVS (*www.viewvc.org/*) para poder examinar vía web el árbol del proyecto.

5.3.2. Interfaz de alertas

Como punto de partida del proyecto, se ha programado en PHP un interfaz web para la visualización de las alertas en la base de datos. Desde dicho interfaz es posible ordenar las alertas, filtrarlas según el impacto que puedan tener en el sistema, y ver un detalle completo de cada una de ellas. Además, se ha instalado PhpPgAdmin (*http://phppgadmin.sourceforge.net/*) para poder acceder y administrar la base de datos de forma cómoda desde el interfaz de alertas.

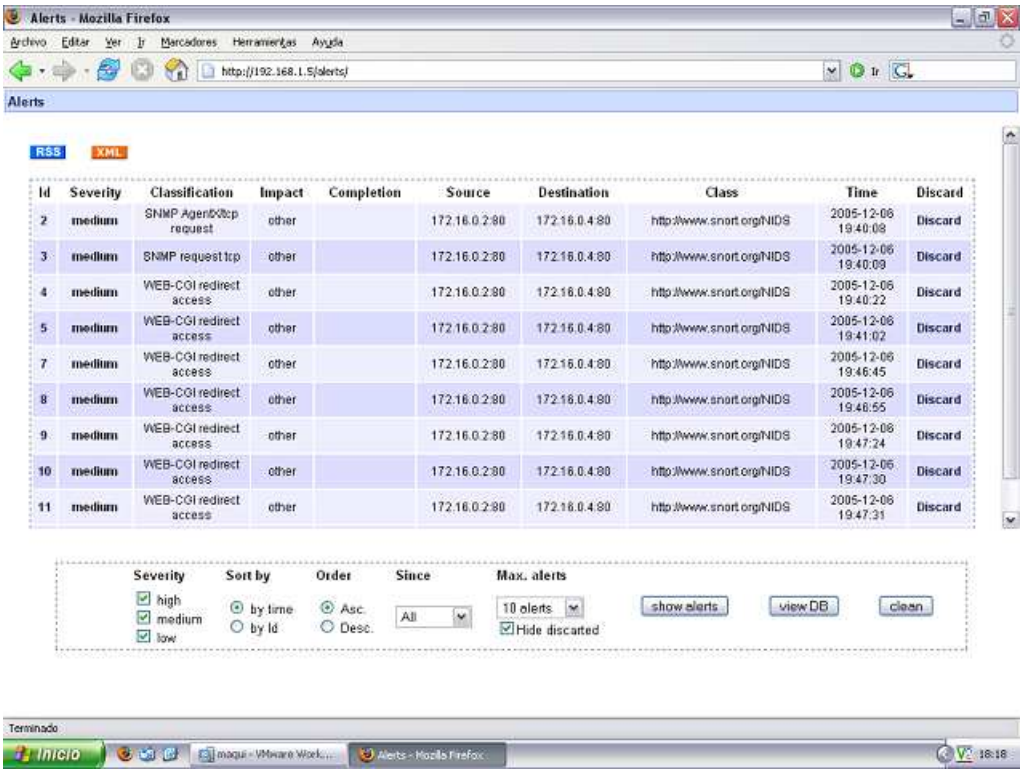


Figura 5.3: Interfaz de alertas

Hemos modificado también la estructura de la base de datos generada por la librería *libpreludedb0* para poder descartar alertas de cara a la visualización. Hemos añadido una columna *hidden* que tomará un valor booleano a la tabla *prelude_alert*, que es la tabla en que se guarda el identificador de la alerta y que sirve de raíz para obtener del resto de tablas la información relacionada con dicha alerta.

Generador de feeds RSS

En el interfaz de alertas se ha añadido un botón para generar al vuelo un feed RSS con las alertas deseadas. El generador lee las alertas de la base de datos en base a unos parámetros dados, y crea un fichero en formato XML con las alertas correctamente for-

mateadas según la especificación de RSS. La URL de este generador es la que utilizarán los clientes suscriptores que deseen recibir las alertas publicadas en un entorno local, para distribuirlas posteriormente en la infraestructura publicador/suscriptor.

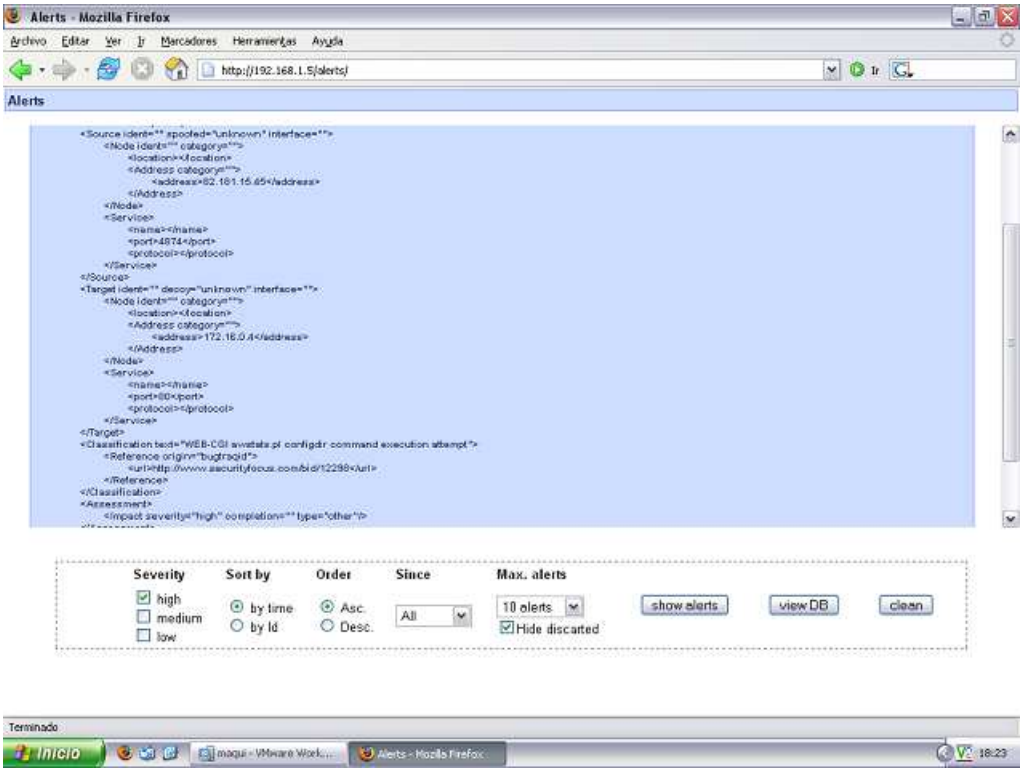


Figura 5.4: Detalles de una alerta en formato IDMEF

Debido a que las alertas incluidas en el feed RSS las formateamos en HTML para una cómoda visualización en aplicaciones agregadoras de feeds comunes, para facilitar el desarrollo posterior de un agregador de feeds RSS capaz de parsear el documento en busca de alertas en formato IDMEF, se guardará dentro de un comentario XML, la alerta original sin formatear. De este modo el agregador que implementaremos para descargar los feeds de alertas y publicarlos en el segundo sistema publicador/suscriptor no tendrá que saber cómo están formateadas las alertas dentro del feed.

5.3.3. Agregador de feeds RSS

El agregador de feeds que se ha programado para descargar las alertas de la base de datos local, se ha escrito en C. Se ha elegido este lenguaje y no uno de más alto nivel, que seguramente permitiera resolver el problema de una manera más sencilla, porque una vez descargadas las alertas, ha de verificar que realmente sean alertas IDMEF bien formadas; de lo contrario no se publicarán en la red de comunicaciones. Para ello hemos utilizado la

librería *libidmef*.

Esta librería nos proporciona la funcionalidad necesaria para crear nuevas alertas en formato IDMEF y para leer el contenido de un fichero o de un buffer y parsearlo para formar variables (la librería no utiliza la orientación a objetos) que representan la alerta IDMEF y poder tratarla convenientemente.

La función principal del agregador de feeds es descargar un feed a partir de una URL, parsearlo en busca de las alertas IDMEF, y guardar las alertas encontradas en un fichero en disco para su posterior tratamiento. El rendimiento del agregador (asumiendo que siempre será utilizado para publicar las alertas en la infraestructura de comunicación) podría verse mejorado si se fueran publicando las alertas a medida que se van encontrando en el feed, pero esto supone un incremento de la complejidad, y que no habría diferenciación entre un cliente publicador de mensajes en xmlBlaster y el cliente suscriptor de alertas basado en canales. Como esto no es un punto excesivamente importante para el objetivo del proyecto, se ha optado por almacenar las alertas encontradas en un fichero temporal.

El agregador de feeds proporciona la siguiente función para descargar un feed de Internet:

```
/* Download and save feed to disk */
int getFeed(int *sock,
            char *host, char *url, char *auth,
        char *fname);
```

Los parámetros que recibe son:

- *int *sock* - El socket que mantiene la conexión con la URL correspondiente al generador de feeds.
- *char *host* - El nombre de dominio de la máquina del agregador de feeds.
- *char *url* - La URL completa para descargar el feed de alertas.
- *char *auth* - La autenticación en caso de que la url del generador de feeds esté protegida.
- *char *fname* - Nombre del fichero en el que se guardarán las alertas descargadas

Para descargar el feed RSS se construye primero una consulta HTTP, y en ella se codifica (en caso de que la haya) la autenticación en *base64* utilizando la librería *OpenSSL*. Seguidamente se envía la petición HTTP generada y se parsea la respuesta obtenida en busca de alertas en formato IDMEF que se almacenarán en un fichero XML en disco. El fichero generado tiene el siguiente formato:

```
<IDMEF-Message-List>
    <IDMEF-Message version=.....>
    </IDMEF-Message>
    ....
    <IDMEF-Message version=.....>
    </IDMEF-Message>
</IDMEF-Message-List>
```

Para que el generador de feeds pueda ejecutarse periódicamente, se le ha añadido un parámetro que permite especificar el intervalo de tiempo tras el cual ha de volver a descargarse el feed de forma automática, y un parámetro que permite ejecutar el agregador de feeds en segundo plano.

Debido a la cantidad de parámetros que puede recibir el feed, y que éstos pueden ser ampliados en un futuro para acotar más los criterios de las alertas que se descargarán mediante este sistema, se ha creado un pequeño parser de ficheros de configuración. Así será posible definir todos los parámetros en un único fichero de configuración (debidamente comentado) y tener una configuración y un código más limpio.

Las variables y funciones principales del parser de ficheros de configuración son:

```
struct _option
{
    char option[50];
    char value[50];
};

typedef struct _option Option;
typedef Option* OptionPtr;
typedef OptionPtr* OptionArr;

OptionArr read_config(char* filename);
void free_options(OptionArr opts);
```

Con la función *read_config* podemos leer el fichero de configuración indicado y obtener un array de punteros a estructuras del tipo *Option*, que contienen el nombre de la opción y su valor. Un ejemplo de fichero de configuración para el agregador de feeds sería el siguiente:

```
#
# Feed Agregator configuration file.
#
# Options must be in the following format:
#   option = value
```

```
#
# Lines beginning with '#' are comments
#

# Execution mode
  mode = normal                    # Mode: normal | daemon

# Agregator parameters
  url = http://localhost/alerts/php/genRSS.php
  output = alerts.xml

  login = pfc                      # Login
  passwd = proyecto                # Password

  max = 10              # Maximum number of feeds to download
  update = 0            # Update interval (0 => don't update)
```

Publicación en el subsistema de comunicaciones

Para publicar las alertas descargadas en la infraestructura publicador/suscriptor global, se ha programado una pequeña aplicación que utiliza el agregador de feeds para descargarse las alertas, parsea el fichero XML generado por el agregador en busca de las alertas individuales, y las publica en los encaminadores xmlBlaster.

Para el parseo del fichero de alertas generado por el agregador de feeds, se ha utilizado la librería *libxml* (*The XML C parser and toolkit of Gnome*). Esta librería nos proporciona la funcionalidad necesaria para manipular documentos XML tanto a partir de su estructura DOM, como utilizando expresiones XPath. En nuestro proyecto hemos utilizado los filtros XPath para obtener por separado todas las alertas y publicarlas en la infraestructura de comunicaciones. La manera en que se publican las alertas se explica en el siguiente apartado.

5.3.4. Interfaz publicador/suscriptor para xmlBlaster

La necesidad de publicar y suscribirse a diferentes subconjuntos de alertas es una necesidad básica que tendrán todos los componentes del sistema que necesiten divulgar o recibir información. Tanto si son analizadores, como managers, o cualquier otro elemento, necesitan tener una metodología común para efectuar las operaciones de publicación y suscripción. Por esto se ha desarrollado un interfaz que permite encapsular la interacción con encaminadores xmlBlaster y que proporciona la funcionalidad necesaria para realizar

las operaciones comunes con los mensajes que se quieren divulgar.

Por cuestiones de flexibilidad, no se ha encapsulado en dicha interfaz la estructura del mensaje que utiliza xmlBlaster (recordemos que se compone de tres partes: key, payload y value). Los diferentes campos pueden transportar información muy variada. El ejemplo más claro es el campo QoS, que podemos utilizar para muchos propósitos diferentes dependiendo del tipo de mensaje.

La interfaz que se ha desarrollado, se ha compilado en forma de librería de enlace dinámico, ya que únicamente pretende ser una abstracción de la lógica de xmlBlaster, y consta de funciones que serán comunes a todos los elementos de la plataforma que deseen comunicarse con los encaminadores. Consideramos que no es necesario que cada elemento de la plataforma tenga una copia de dicho código, y por tanto hemos desarrollado esta parte como una librería de enlace dinámico.

La interfaz proporciona las siguientes funciones:

```
/* Publish methods */
enum p_type { NORMAL, ONEWAY, ARRAY };

/* xmlBlaster connection variables */
extern XmlBlasterException xe;
extern XmlBlasterAccessUnparsed* xa;

/* Connection functions */
void psInit(int argc, char** argv, UpdateFp myUpdate);
void psConnect(char* callbackSsessionId,
               char* user, char* pass, UpdateFp myUpdate);
void psDisconnect();

/* Publish functions */
void psPublish(char* key, char* content, char* qos,
               enum p_type type);

/* Subscribe functions */
void psSubscribe(char* key, char* qos);
void psUnsubscribe(char* key, char* qos);

/* Data management functions */
void psErase(char* key, char* qos);
void psSynchronize(char* key, char* qos);
```

Como puede apreciarse, la interfaz nos define los diferentes métodos de publicación que pueden ser empleados por los clientes publicadores, las variables necesarias para

mantener el estado de la conexión entre el cliente y el encaminador xmlBlaster y para estar informados de los errores que puedan ocurrir, y las funciones que utilizarán los clientes de la infraestructura publicador/suscriptor para divulgar sus mensajes y acceder la información requerida.

A continuación presentaremos brevemente las funciones del interfaz:

- **void psInit(int argc, char** argv, UpdateFp myUpdate)** - Esta función es la que inicializa las variables necesarias para mantener la conexión con xmlBlaster. Recibe los argumentos propios de la función principal de una aplicación para pasárselos al encaminador xmlBlaster, y un tercer parámetro *myUpdate* que corresponde a la función de actualización. Los clientes suscriptores deberán implementar dicha función que será llamada cada vez que se reciba un mensaje. Esta función puede utilizarse para visualizar el mensaje recibido, generar una respuesta para dar al encaminador, etc. En el caso de los clientes publicadores, se dejará este parámetro a *NULL*.

- **void psConnect(char* callbackSsessionId, char* user, char* pass, UpdateFp myUpdate)** - Esta función es la que establece la conexión del cliente con el encaminador xmlBlaster. Los parámetros que recibe son el *callbackSsessionId*, que es un identificador de la sesión que utilizará xmlBlaster para saber a qué cliente ha de enviar los mensajes de respuesta. Recibe también un usuario y una contraseña (estos parámetros serán discutidos más adelante), y una función de actualización como la de la función anterior en el caso de clientes suscriptores.

- **void psDisconnect()** - Esta función sirve para finalizar la conexión con el encaminador xmlBlaster.

- **void psPublish(char* key, char* content, char* qos, enum p_type type)** - Con esta función pueden publicarse mensajes en un encaminador xmlBlaster. Los parámetros que recibe son cadenas de caracteres (en formato XML) que representan los campos básicos que forman los mensajes de xmlBlaster. En nuestra plataforma, el parámetro *content* contendrá la alerta en formato IDMEF que se quiere publicar. El último parámetro determina el método de publicación:

  - *NORMAL*: Se utilizará para publicar los mensajes de forma normal. Será lo habitual.
  - *ONEWAY*: Se usará para publicar los mensajes en un único sentido.
  - *ARRAY*: Se utilizará para publicar de golpe un array de mensajes

- **void psSubscribe(char* key, char* qos)** - Esta función se utilizará para realizar una nueva suscripción a un subconjunto de alertas, especificando los criterios en el campo *key* y los filtros sobre el contenido en el campo *QoS*.
- **void psUnsubscribe(char* key, char* qos)** - Utilizaremos esta función para cancelar la suscripción a un subconjunto de alertas en el momento deseado.
- **void psErase(char* key, char* qos)** - Mediante esta función podemos eliminar del encaminador xmlBlaster un subconjunto de mensajes. Es interesante cuando se desean descartar mensajes que se habían publicado como persistentes.
- **void psSynchronize(char* key, char* qos)** - Esta función permite descargar de forma síncrona el subconjunto de mensajes definidos por los filtros pasados como parámetro.

Por el momento, el método de autenticación utilizado en la función de conexión al encaminador es *htpasswd*. Este método se basa en un fichero con un listado de usuarios y contraseñas codificadas mediante la función *crypt* (como en el fichero */etc/passwd* de los sistemas UNIX) a los que se les permite el acceso a xmlBlaster. La distribución de xmlBlaster proporciona también una herramienta para encriptar las contraseñas mediante este método.

Para configurar la autenticación hay que generar un fichero con los usuarios, contraseñas cifradas, y permisos que tendrá cada usuario en el sistema de la forma:
<usuario>:<contraseña cifrada>:<permisos>.

A continuación podemos ver un ejemplo de dicho fichero:

```
guest:yZ24stvIel1j6:connect,disconnect,subscribe
pfc:yZ24stvIel1j6:connect,disconnect,publish(alert),subscribe(alert)
nacx:yZ24stvIel1j6:subscribe(.exact:alert";"xpath://key[starts-with(@oid,'high.')]")
admin:yZ24stvIel1j6:!erase
```

En este ejemplo se han definido los siguientes usuarios y permisos:

- El usuario **guest** podrá conectarse, desconectarse y suscribirse a un subconjunto de mensajes en el encaminador.
- El usuario **pfc** podrá conectar, desconectar, publicar únicamente mensajes con el tema *alert*, y suscribirse únicamente a mensajes con el mismo tema.

- El usuario **nacx** podrá suscribirse únicamente a mensajes con el tema *alert* o a aquellos que cumplan el filtro XPath definido.
- El usuario **admin** podrá efectuar todas las operaciones excepto eliminar mensajes.

Una vez generado el fichero, hay que añadirlo tanto para el cliente como para el servidor, en el fichero de configuración de xmlBlaster de la forma siguiente:

```
# Para el servidor:
Security.Server.Plugin[htpasswd][1.0]=\
org.xmlBlaster.authentication.plugins.htpasswd.Manager

Security.Server.Plugin.htpasswd.secretfile=<ruta al fichero de acceso>

# Para los clientes:
Security.Client.Plugin[htpasswd][1.0]=\
org.xmlBlaster.authentication.plugins.htpasswd.ClientPlugin
```

En lo referente a la publicación de mensajes, la función de publicación del interfaz controlará que las alertas que se pretendan publicar estén en un formato IDMEF válido. Para ello se utilizará la librería *libidmef*. En vez de crear un objeto propio de dicha librería que contiene la alerta IDMEF y publicar su volcado en XML en el contenido, utilizaremos la librería únicamente para verificar que el formato de la cadena de caracteres suministrada como contenido sea una alerta IDMEF válida. Esto lo hacemos así porque las funciones de volcado en XML de la librería *libidmef* incluyen en el XML resultante una serie de cabeceras que dan conflicto con el plugin de xmlBlaster *MIME Access XPath* a la hora de aplicar los filtros de contenido definidos con expresiones XPath. Por ello, hemos considerado que es más sencillo verificar con la librería *libidmef* que la alerta esté bien formada y en caso afirmativo enviar la cadena pasada como parámetro, que hacer un post-proceso de parseo sobre el volcado de la alerta que produce la librería para eliminar las cabeceras que crean el conflicto.

Utilizando las funciones de este interfaz podremos diseñar de manera sencilla un conjunto de clientes publicadores y suscriptores que atenderán cada uno a subconjuntos diferentes de alertas para realizar de manera cooperativa las tareas de correlación y identificación de posibles escenarios de ataque. A continuación presentamos un par de ejemplos de suscripciones a un subconjunto de mensajes:

```
/* Suscribirse a todas las alertas con un impacto alto */
```

```
psSubscribe(
  "<key oid='Alert'/>",
  "<qos>"
  "  <filter type='XPathFilter'>"
  "   //Impact[@severity='high']"
  "   </filter>"
  "</qos>");

/* Suscribirse a las alertas de cuya */
/* IP de origen sea: 24.77.190.99 */
psSubscribe(
    "<key oid='Alert'/>",
    "<qos>"
    "  <filter type='XPathFilter'>"
    "     //Source//address/text()='24.77.190.99'"
    "   </filter>"
    "</qos>");
```

Es importante remarcar que siempre que utilicemos el plugin *MIME Access XPath* para realizar filtros sobre el contenido, en los mensajes publicados deberemos especificar en el campo key el parámetro *contentMime='text/xml'*, ya que dicho plugin sólo está disponible para este tipo de datos en el campo *payload* de los mensajes xmlBlaster. Un ejemplo de campo *key* de un cliente publicador sería:

```
psPublish(
    "<key oid='Alert' contentMime='text/xml'/>",
    "<IDMEF-Message> ... </IDMEF-Message>",
    "<qos>"
    "     <persistent/>"
    "</qos>",
    NORMAL);
```

Notas sobre la organización interna de los mensajes

Es conveniente aclarar un punto de la publicación de mensajes con contenido adicional en el campo key (para hacer éste más específico y poder realizar sobre este un primer filtrado mediante consultas XPath sobre esta parte del mensaje) que puede inducir a ciertos errores a la hora de diseñar la suscripción a un conjunto de mensajes.

Cuando un mensaje llega al encaminador, éste analiza en primera instancia el atributo **oid** (o tema) del campo *key*, y ningún otro, para comprobar si el tema existe o es un tema nuevo. Si el tema no existe hasta el momento, xmlBlaster crea internamente uno

nuevo con las propiedades del campo key del mensaje recibido; tanto el *oid* como el resto del contenido de dicho campo (la información adicional que se ha proporcionado con el fin de hacer suscripciones más precisas) conformarán el nuevo tipo de mensajes en el encaminador. En cambio, si el oid del mensaje recibido (el tema) ya existe, **no** se comprueba el resto del campo key. Aunque la información adicional que posea sea diferente de la información adicional del tema que ya existía con el mismo oid, ésta no es analizada y el mensaje se clasifica en una estructura de árbol como hijo del ya existente, por lo que desde el punto de vista del encaminador, este último mensaje comparte tema con su padre, y por tanto los filtros XPath o Regexp sobre el campo key serán aplicados a la información adicional del padre y no del hijo. Es decir, los mensajes se clasifican internamente **únicamente** por temas (por el atributo *oid*); si dos mensajes comparten el mismo tema pero tienen contenido adicional distinto, prevalecerá el contenido adicional del mensaje que ya existiera en el encaminador.

Es importante remarcar esta característica, ya que los filtros XPath sobre el campo *key* del mensaje nos serán útiles para obtener un subconjunto de mensajes que sean de temas distintos, pero no para obtener un subconjunto de mensajes que compartan un mismo tema.

Esta clasificación interna que realiza xmlBlaster nos limita a la hora de definir los filtros que queremos aplicar sobre los mensajes. Por ello, si queremos realizar consultas específicas utilizamos el plugin *MIME Access XPath* para definir filtros sobre el contenido del mensaje. Con el uso de este plugin en las suscripciones, se obtendrán primero del encaminador todos los mensajes que se correspondan con el tema especificado en el campo key, y luego se aplicarán los filtros definidos en la configuración del plugin sobre el subconjunto de mensajes resultantes de la primera consulta al encaminador.

De este modo podemos tener una manera eficiente de clasificar y realizar la suscripción al subconjunto de alertas deseado. Podemos tener en el campo key un primer elemento discriminante, y luego aplicar un filtrado más específico sobre el conjunto resultante de este filtrado, optimizando el rendimiento al reservar el trabajo más pesado a un conjunto más pequeño de mensajes.

Es evidente que el modo en que diseñemos los temas y los filtros aplicables sobre el payload condicionarán la manera en que xmlBlaster clasificará internamente los mensajes, la manera en que podremos suscribirnos, y el rendimiento final del sistema. Es importante, por lo tanto, el estudio de las diferentes formas de publicar mensajes y el rendimiento que ofrece el sistema frente a las diferentes suscripciones que pueden realizarse, con el fin de tener una metodología común y eficiente de publicación de mensajes en los diferentes encaminadores del subsistema de comunicaciones.

5.3.5. Publicación mediante triggers

Para implementar el segundo modelo de comunicaciones completamente asíncrono, necesitamos algún elemento que publique en el sistema de comunicación las alertas a medida que son generadas. En nuestra plataforma, esto podría hacerse programando extendiendo o programando nuevos managers de Prelude que a medida que vayan detectando las alertas las publiquen, o bien mediante triggers que se disparen ante la inserción de nueva información en la base de datos de alertas.

La primera opción es bastante más costosa, requiere un estudio previo de la API propia de Prelude, y las ventajas que aporta sobre los triggers (teniendo en cuenta que lo único que se pretende es difundir la alerta) no son considerables. Optaremos entonces por la implementación de triggers de PostgreSQL para la divulgación de alertas.

Para programar el trigger se ha utilizado la SPI (*Server Programming Interface*) de PostgreSQL y su API para el desarrollo de triggers en C. Utilizando estas librerías se desarrolló un primer trigger que escribía la información relacionada con una alerta en un fichero en disco. El trigger debe compilarse como una librería de enlace dinámico, aunque no es necesario especificarle su ubicación al enlazador del sistema; únicamente se necesita la librería para importarla en la base de datos PostgreSQL. La importación de la librería y creación del trigger se haría del siguiente modo:

```
CREATE FUNCTION read_alert() RETURNS trigger
    AS '/home/pfc/dblistener/libdbalert.so.1.0' LANGUAGE C;

CREATE TRIGGER tafter AFTER INSERT ON prelude_alert
    FOR EACH ROW EXECUTE PROCEDURE read_alert();
```

Donde */home/pfc/dblistener/libdbalert.so.1.0* es la librería dinámica que hemos compilado, y *read_alert()* es la función del trigger que se encargará de leer la información recién insertada en la base de datos.

El desarrollo del trigger, a pesar de ser a priori una solución buena y no excesivamente complicada, presenta una serie de limitaciones y complicaciones:

- No pueden usarse nombres de funciones en la librería que coincidan con símbolos internos de PostgreSQL. Si así lo hacemos, el SGDB no será capaz de cargar la librería y nos dará un error del estilo: *ERROR: no se pudo cargar la biblioteca <libreria>undefined symbol: <función>*

- Al hacer las consultas necesarias contra la base de datos para recoger la información de las diferentes tablas en las que se guarda información de la alerta, algunas no

devuelven ningún resultado. Esto puede ser debido a que el trigger lo disparamos cuando se inserta un nuevo elemento en la tabla *prelude_alert* (que es el instante en el que se genera una alerta) pero ejecuta las consultas antes de que el SGDB haya podido insertar la información en el resto de tablas.

El primer inconveniente se evita fácilmente eligiendo bien los nombres que tendrán las variables y funciones de la librería dinámica que implementa el trigger. El segundo inconveniente, en cambio, hemos tratado de solucionarlo retardando la ejecución de las consultas sobre terceras tablas, incluso haciendo nuevos triggers para dichas tablas. En el caso del retraso el resultado es el mismo, y el caso de los triggers no lo consideramos una buena solución; puede ser muy costoso de mantener, no es escalable, y requiere un post-proceso para construir de manera correcta la alerta en formato IDMEF a partir de la información generada por diferentes triggers.

Por estas razones, y debido a que el desarrollo de triggers eficientes no es un parámetro que vaya a ser discriminatorio a la hora de evaluar el sistema de comunicaciones de la plataforma, hemos desestimado la utilización de estos triggers de PostgreSQL para el proyecto actual, en favor de la publicación de alertas local basada en canales comentada en apartados anteriores.

5.3.6. Configuración de entornos de cluster

Una de las principales características que nos ofrece xmlBlaster es la configuración de entornos de cluster, mediante la cual una serie de encaminadores son configurados para compartir información en base a unas reglas encaminamiento basadas en un modelo maestro/esclavo. Estas reglas pueden definirse mediante la definición de diferentes dominios (utilizando el atributo *domain* en el campo key de los mensajes), mediante filtros sobre el campo key (por *oid* o mediante XPath) o mediante los filtros propios del campo QoS.

Durante el desarrollo del proyecto se han llevado a cabo diferentes configuraciones y se ha podido comprobar que en la mayoría de ellas el comportamiento es el esperado. Aún así, se han experimentado problemas a la hora de utilizar el encaminamiento basado en dominios en alguna de las configuraciones que se comentarán a continuación.

Configuración básica

La configuración del entorno de cluster se ha hecho de manera incremental. Primeramente se ha realizado una configuración de cluster básica entre dos nodos y reglas de encaminamiento basadas en dominios. Para ello, se ha configurado un nodo como maestro para los mensajes del dominio *Alert*, y el segundo como esclavo de éste.

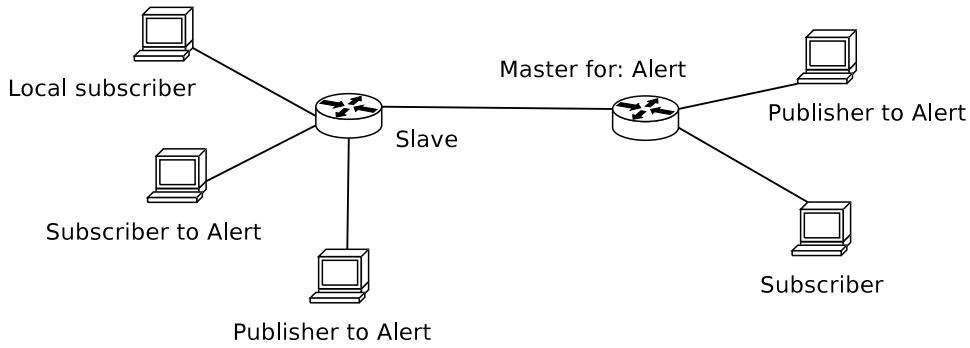


Figura 5.5: Configuración simple de un cluster

Las alertas publicadas en el dominio *Alert* serán enviadas al nodo maestro. Los clientes suscriptores de dicho dominio que estén directamente conectados al maestro, realizarán su suscripción de forma local y recibirán las alertas. Los clientes conectados de forma local al esclavo, enviarán a éste su suscripción indicando que es para el dominio *Alert* y éste la redirigirá al nodo maestro para que sepa a través de qué nodo ha de entregar las notificaciones correspondientes a dichas suscripciones.

Configuración de tres nodos y tres dominios

Una vez comprobado el correcto funcionamiento de la configuración anterior, se añadió un tercer encaminador en el cluster y se configuraron tres dominios distintos en función del la severidad de las alertas: alta, media y baja. Se configuró cada uno de los nodos para ser maestro para los mensajes de una severidad, y esclavo para los mensajes de las otras dos. Esta configuración es prácticamente idéntica a la del caso anterior, con la salvedad de que ahora existe un encaminador más, y que hay definidos tres dominios en vez de uno; sin embargo aparecieron problemas en cuanto a la recepción de alertas en los nodos de los distintos dominios.

Tras la revisión de las trazas y la información de debug del plugin de clustering de los encaminadores, se pudo apreciar cómo las alertas eran distribuidas de forma correcta hasta los maestros de los diferentes dominios definidos. En dichas trazas se pudo apreciar claramente cómo cada uno de los encaminadores resolvía a qué encaminador debía reenviar el mensaje en función de las reglas de encaminamiento definidas. El problema se presenta en los clientes suscriptores de alertas publicadas en dominios definidos en encaminadores remotos. Desde un cliente suscriptor de un dominio en un encaminador local, se recibían correctamente las alertas; en cambio, desde un cliente suscriptor de un dominio definido en un encaminador remoto se recibía únicamente una alerta. Indepen-

dientemente de cómo estuviera formado el mensaje y del campo *key* que se le hubiera especificado, siempre se recibía una única alerta.

Podemos descartar un problema en la definición de las reglas de encaminamiento entre los nodos diferentes nodos de la infraestructura de comunicación, ya que la publicación de mensajes se pudo hacer de forma correcta en todo momento, y en las trazas e información de debug del plugin de clustering se puede apreciar el intercambio de mensajes de *keepalive* entre los diferentes nodos del cluster. Podemos descartar también una malformación de los mensajes de suscripción, ya que la alerta que se recibe corresponde con la suscripción efectuada. También podemos descartar una malformación en los mensajes publicados, ya que el contenido de todos los mensajes es validado previamente utilizando la librería *libIDMEF* [3], y los campos *key* y *qos* son generados al vuelo por el programa publicador en función de la severidad del mensaje. El hecho de que todos los mensajes sean tratados del mismo modo, y que para todos ellos se reciban las correspondientes confirmaciones del encaminador conforme han sido publicados correctamente, hacen que se pueda descartar la posibilidad de que los mensajes publicados estén mal formados. Tomando en consideración todas estas observaciones, y la información obtenida de las trazas del plugin de clustering, nos inclinamos a pensar en un posible fallo en la selección de mensajes de un dominio en concreto por parte de xmlBlaster.

Cluster de tres nodos basado en filtros

En vista de que las reglas de encaminamiento basadas en dominios suponían un problema, se decidió configurar el encaminamiento en base a filtros sobre el campo key. Así, configuramos los publicadores para que publicaran en el tema del mensaje el nivel de severidad de la alerta (alto, medio o bajo); y los encaminadores con filtros que los definían como maestros de los mensajes cuyo tema fuera, también, alto, medio o bajo. De este modo se pudo omitir el uso de dominios.

En la figura 5.6 puede verse el esquema resultante. Los mensajes publicados son reenviados a su nodo maestro en base a su severidad, y como ocurría en el primer modelo de clustering presentado, las suscripciones a una severidad concreta son dirigidas también al nodo maestro para que éste sepa el camino de vuelta para realizar al entrega de mensajes a los clientes suscriptores.

Este esquema se comporta de la forma esperada, pero presenta el inconveniente de perder la libertad a la hora de elegir el tema de los mensajes. No sólo los publicadores deben utilizar un tema específico, sino que los suscriptores sólo pueden suscribirse a un tema específico, lo que los limita a recibir únicamente alertas de una severidad concreta, y

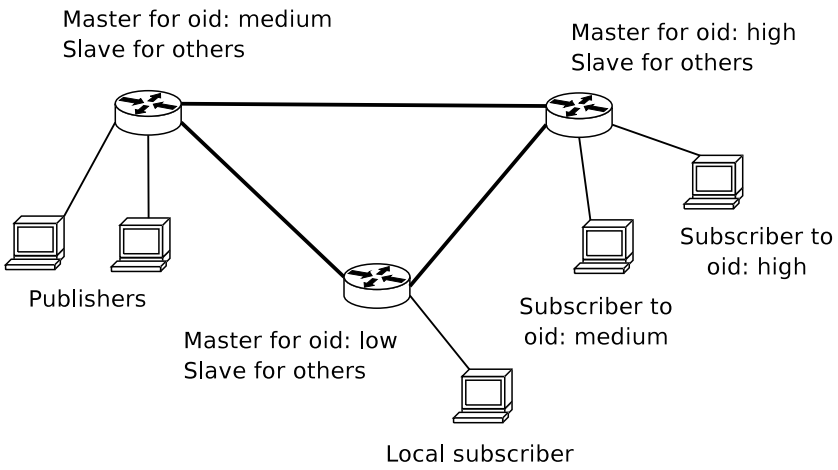


Figura 5.6: Cluster de tres nodos basado en filtros

hace necesaria la utilización de múltiples suscriptores para recibir un abanico más amplio. No obstante, este comportamiento puede mejorarse mediante la adopción de filtros XPath para el establecimiento de los nodos maestros y las suscripciones de los mensajes. Una posible configuración sería la publicación de alertas bajo el tema: *Alert-<severidad>*. De este modo, el tema tendría una parte común y conocida previamente, y una parte variable (el severidad). A partir de ahí se pueden definirse los tres nodos maestros para los mensajes de los temas: *Alert-alta*, *Alert-media* y *Alert-baja*, y los clientes suscriptores pueden suscribirse a las alertas mediante filtros XPath sobre el campo key de la forma:

```
<key queryType='XPATH'>//key[starts-with(at oid,'Alert')]</key>
<key queryType='XPATH'>//key[@oid='Alert-alta']</key>
<key queryType='XPATH'>
   //key[@oid='Alert-alta'] | //key[@oid='Alert-media']
</key>
```

De este modo se resuelve el problema de la limitación de los nodos suscriptores a los mensajes de un único tema, y pueden definirse los maestros de los mensajes en base a diferentes criterios referentes a las alertas.

5.4. Resumen

En este capítulo hemos visto los diferentes componentes que conforman la plataforma de detección de intrusiones, los requisitos previos del sistema implementarla, y las aplicaciones y librerías que se han desarrollado para que puedan comunicarse entre sí utilizando el sistema publicador/suscriptor. También se ha abordado las diferentes configuraciones de los entornos de clustering que se han empleado para evaluar la plataforma,

las características de cada una de ellas, los problemas aparecidos y las soluciones que se han adoptado para resolverlos.

Capítulo 6

Evaluación de la plataforma

En este capítulo presentamos la evaluación de la plataforma que implementa la infraestructura de comunicaciones desarrollada en este proyecto. Para la evaluación, se han realizado las pruebas que se detallan en los siguientes apartados. Las pruebas fueron realizadas en un equipo con procesador Intel Pentium 4 a 1.5GHz, con 256 MB de memoria RAM, funcionando sobre un sistema operativo Debian GNU/Linux 2.6, configurado con un servidor de HTTP Apache 1.3, un intérprete de PHP/4.3 y un cliente Java HotSpot VM 1.5 (para la ejecución de los encaminadores xmlBlaster).

La generación de alertas en formato IDMEF se ha realizado a partir de un conjunto de los analizadores gestionados por Prelude de que consta la plataforma (entre ellos Snort). Para este estudio, se han descargado las alertas en formato IDMEF de la base de datos utilizando el agregador de feeds RSS implementado, se han parseado las alertas necesarias, y se han publicado las alertas en la red de encaminadores xmlBlaster utilizando las funciones de la interfaz desarrollada, para que diferentes clientes suscriptores obtengan los mensajes deseados.

6.1. Consumo de CPU y espacio de memoria

El consumo de CPU y espacio de memoria utilizado se ha realizado por separado para los encaminadores xmlBlaster, los clientes publicadores, y los clientes suscriptores.

6.1.1. Consumo de los encaminadores

Para las pruebas de consumo se ha evaluado el rendimiento de la plataforma ante la publicación de hasta diez mil alertas de golpe. El consumo medio de recursos por parte de los encaminadores xmlBlaster y los intervalos de confianza definidos para cada valor medio pueden verse en la figura 6.1.

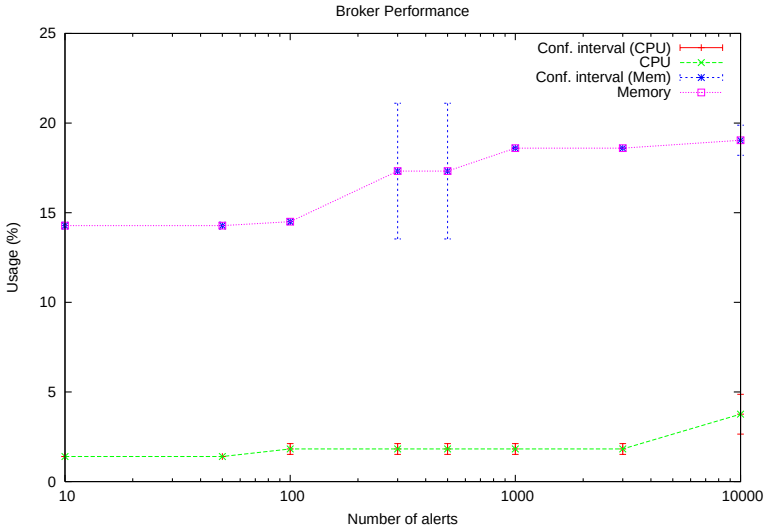


Figura 6.1: Consumo medio de los encaminadores

Puede apreciarse que tanto el consumo de CPU como de memoria se mantiene bastante estable durante todo el proceso (alrededor del 16 % y el 4 % respectivamente), experimentando un ligero incremento al crecer el número de alertas recibidas. Al incrementarse el número de alertas, el consumo experimenta una ligera subida, pero siguiendo una línea bastante estable en todo momento.

6.1.2. Consumo de los publicadores

En la figura 6.2 puede apreciarse el rendimiento de los clientes publicadores. Hay que tener en cuenta, que estos publicadores, tal y como los hemos implementado en esta plataforma, parsean un fichero de alertas previamente descargado mediante el agregador de feeds y publican las alertas encontradas en dicho fichero.

Como se puede apreciar, el consumo de recursos crece de forma bastante suave hasta alrededor de las 3000 alertas publicadas. Cuando el fichero contienen más de 3000 alertas, el consumo de recursos se dispara. Consideramos que este incremento de consumo en los publicadores se debe al análisis del fichero que contiene las alertas en formato IDMEF más que a las funciones propias de publicación y suscripción. Este fichero puede ocupar un espacio considerable en disco y su parseo (que en nuestros publicadores hacemos mediante filtros XPath) bastante costoso.

6.1.3. Consumo de los suscriptores

En la figura 6.3 se muestran los recursos consumidos por los clientes suscriptores.

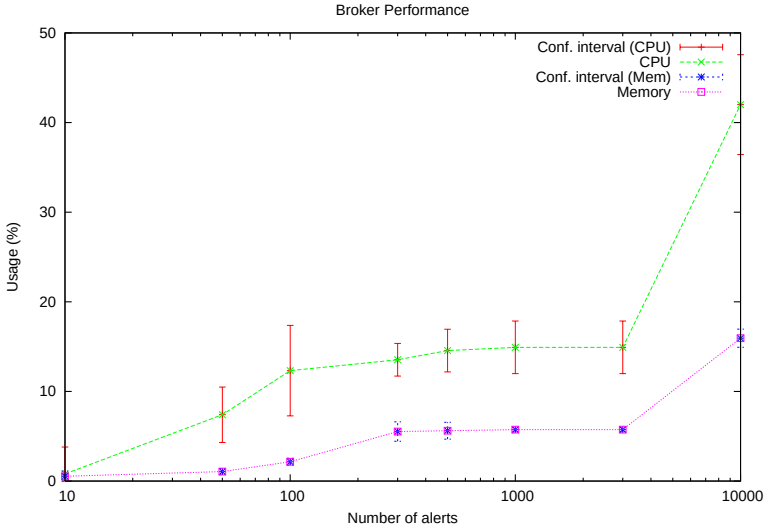


Figura 6.2: Consumo medio de los publicadores

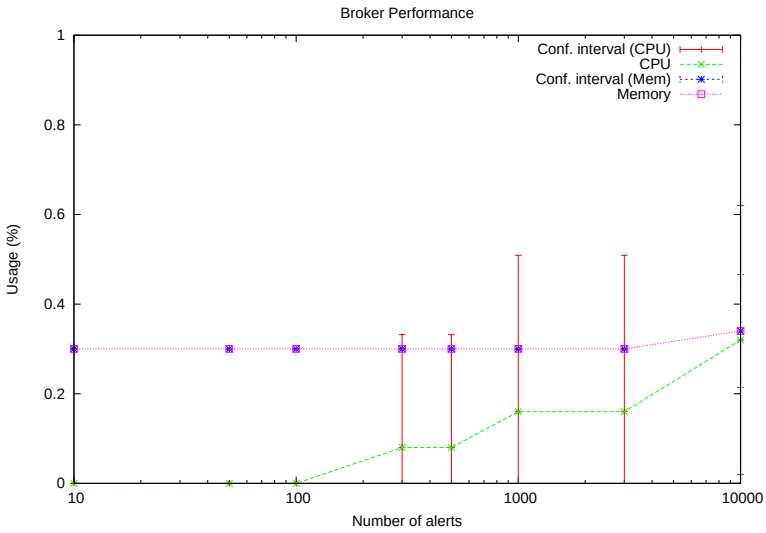


Figura 6.3: Consumo medio de los suscriptores

Como puede verse en la figura, el consumo de CPU se mantiene estable independientemente de número de alertas, alrededor del 0.3 %, y el consumo de memoria experimenta un ligero incremento a medida que crece el número de alertas publicadas, pero sin llegar a superar el 0.5 %.

Por ello, concluimos que en el caso de utilizar esta plataforma para la comunicación de alertas en un escenario de detección real, el consumo de memoria y CPU por parte de encaminadores y clientes suscriptores se mantendrá constante, independientemente del número de mensajes IDMEF publicados, mientras que el consumo de CPU por parte de los clientes publicadores dependerá del análisis y generación de los mensajes a publicar.

6.2. Rendimiento en la entrega de notificaciones

En esta segunda prueba se ha analizado la latencia de notificación por parte de los encaminadores xmlBlaster, publicando alertas organizadas de tres maneras diferentes: una primera manera en la que todos los mensajes compartirán un mismo tema, implicando que internamente xmlblaster almacenará estos mensajes en un árbol bajo un mismo nodo raíz; una segunda manera organizando los mensajes en tres temas distintos basados en el impacto de la alerta (alto, medio y bajo), que serán organizados por xmlBlaster en un árbol con tres nodos principales de los que colgarán respectivamente los mensajes de cada tema; y una tercera manera en la que cada mensaje pertenecerá a un tema distinto (para ello se ha utilizado como tema el ID de la alerta), haciendo que xmlBlaster tenga un nodo principal para cada mensaje IDMEF.

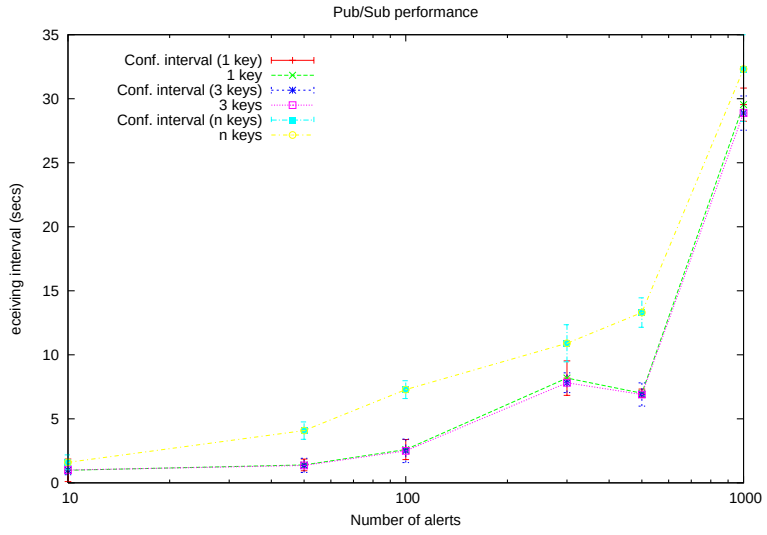


Figura 6.4: Evaluación de la entrega de notificaciones

Analizando el tiempo de entrega en los dos primeros casos de organización de mensajes (ver figura 6.4), se ha podido comprobar que para un número inferior a 100 mensajes, apenas hay diferencias. A partir de ahí, se aprecia que la latencia en la entrega de mensajes mejora ligeramente en el segundo caso de organización a medida que crece el número de mensajes. Creemos que esto se debe a que la búsqueda por parte del algoritmo de encaminamiento de xmlBlaster es más sencilla en el segundo caso al existir una organización en base a tres criterios diferentes.

Por otro lado, al analizar la latencia en el tercer caso de organización de mensajes, el tiempo en la entrega a los suscriptores se resiente ligeramente. De nuevo, consideramos que esto es debido a que el algoritmo de encaminamiento de xmlBlaster se enfrenta a un número de caminos de búsqueda superior, lo que repercute negativamente en la entrega de las notificaciones.

De esta segunda prueba se desprende que la entrega de mensajes IDMEF mediante xmlBlaster no supuso en ningún momento un cuello de botella, pues todos los mensajes fueron procesados y entregados a tiempo, sin alcanzar en ningún momento puntos de saturación. Este resultado nos permite concluir que la utilización de xmlBlaster en un escenario de detección real garantiza las expectativas iniciales, proporcionando la escalabilidad y eficiencia esperadas.

6.3. Rendimiento del agregador de canales RSS

Esta evaluación se ha basado en medir la latencia del agregador de canales RSS a la hora de descargar, analizar y publicar las alertas de la base de datos en xmlBlaster. Los resultados que aparecen en la figura 6.5 muestran como esta latencia depende proporcionalmente del número de alertas a agregar. Como puede verse, a partir de las 1000 alertas, el tiempo de parseo del fichero XML generado por el agregador de canales RSS se dispara. Este es el mismo proceso de parseo que realizan los publicadores, y puede verse que seguramente existan algoritmos de parseo más eficientes que las expresiones XPath. De todos modos, que la manera en que se almacenan las alertas guardadas en disco y el modo en que se parsean los ficheros intermedios sea óptima, queda fuera de los objetivos de este proyecto.

No se creyó necesario tampoco medir los consumos de CPU y memoria del agregador de RSS, ya que este rendimiento depende directamente del servidor de HTTP y del generador de canales seleccionado (Apache 1.3 y PHP/4.3 respectivamente). Consideramos que la utilización de este componente en una aplicación de detección real deberá basarse en un servidor de HTTP más ligero, junto con un gestor de generación de canales RSS

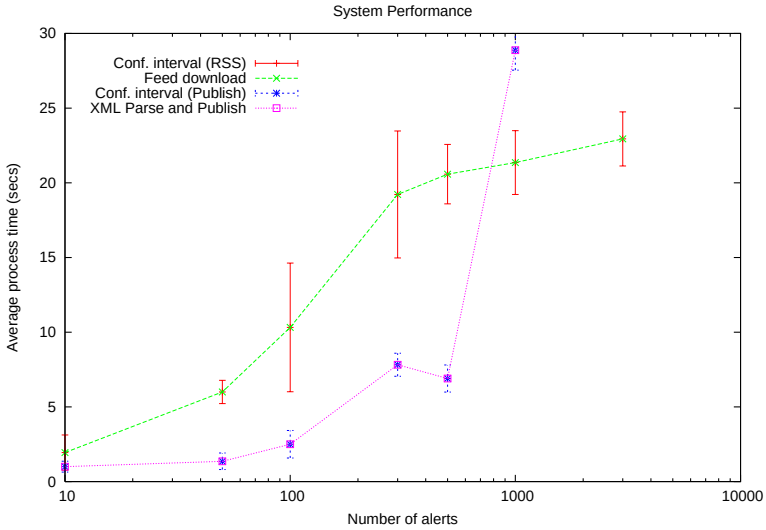


Figura 6.5: Evaluación del agregador de canales RSS

más eficiente.

6.4. Rendimiento del sistema al completo

En esta última prueba se ha evaluado el rendimiento del sistema al completo, a partir de una generación de alertas reales por parte de los analizadores del sistema (gestionados por Prelude) y almacenadas en la base de datos PostgreSQL. Las pruebas se han realizado nuevamente para cada una de las organizaciones de mensajes comentadas en la segunda evaluación.

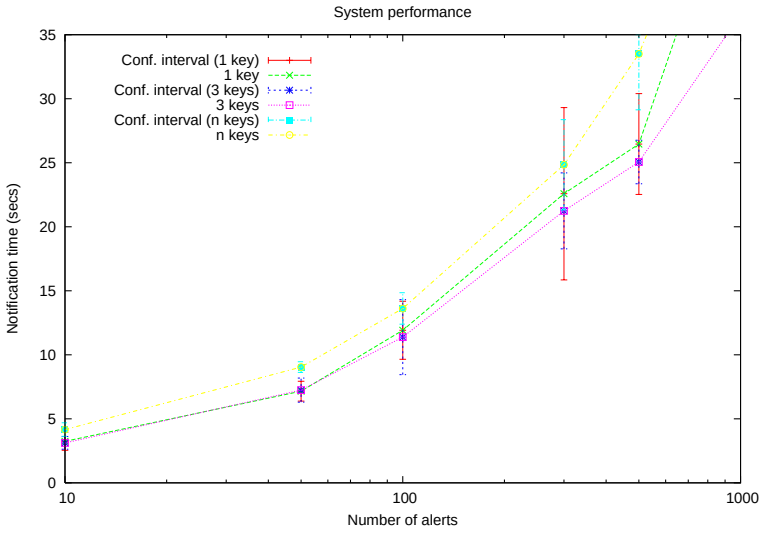


Figura 6.6: Evaluación de la plataforma al completo

Como puede observarse en la figura 6.6, los resultados obtenidos son los esperados tras las pruebas anteriores y siguen apuntando a la segunda organización de mensajes como la mejor opción. también se desprende de esta última prueba que el sistema se comporta de manera estable, siendo el agregador de RSS y el análisis de ficheros XML los elementos que introducen más penalización en el rendimiento del sistema.

6.5. Resumen

En este capítulo hemos visto el estudio llevado a cabo para realizar una evaluación de las diferentes partes que componen la plataforma de detección. Se ha evaluado el rendimiento de encaminadores, publicadores y suscriptores, así como el rendimiento del agregador de canales RSS, y del sistema completo en base a diferentes manera de organizar los mensajes publicados.

Capítulo 7

Conclusiones

En este proyecto se ha presentado una infraestructura para el intercambio de información entre los distintos componentes de una plataforma para la gestión de ataques e intrusiones. Este intercambio de información, realizado a través de mensajes IDMEF, y basado en un modelo publicador/suscriptor, está específicamente diseñado para llevar a cabo de un proceso de correlación de alertas realizado, de forma colaborativa, entre los distintos componentes de nuestra plataforma de detección.

A diferencia de paradigmas tradicionales cliente/servidor para la comunicación de componentes, donde un enfoque centralizado o jerárquico nos lleva rápidamente a problemas de saturación y cuellos de botella, el intercambio de información entre elementos de nuestra plataforma mediante un modelo publicador/suscriptor ofrece una solución eficiente y escalable.

La utilización de las dos tecnologías publicador/suscriptor escogidas para el desarrollo de nuestra infraestructura de comunicación, canales RSS y xmlBlaster, ha sido evaluada sobre el desarrollo de un primer prototipo para sistemas GNU/Linux. Los resultados obtenidos de esta evaluación muestran que el rendimiento de la infraestructura resultante cumple las expectativas iniciales y es el conveniente para realizar el desarrollo completo de la plataforma de detección.

Como líneas de continuidad al trabajo presentado sería interesante la utilización de comunicación segura entre los componentes de la plataforma mediante la utilización de SSL (*Secure Socket Layer*) tanto para la comunicación local, a través de canales RSS, como para la comunicación distribuida entre entornos de detección, a través de xmlBlaster. La utilización de canales de comunicación seguros a través de SSL no tan sólo permitirá la comunicación entre los elementos de la plataforma de forma segura, sino que hará posible la autenticación de componentes que van a interactuar en el algoritmo de detección. Las implicaciones de la inclusión de SSL sobre el actual esquema de comunicación, así como las repercusiones de utilizar una PKI (*Public Key Infrastructure*) en la plataforma, deben

ser aún estudiadas y evaluadas, antes de proceder a su inclusión en el presente trabajo.

Otro aspecto importante a tener en cuenta para un futuro desarrollo es la introducción de técnicas de tolerancia a fallos en la plataforma. Estas técnicas y mecanismos tratarán la gestión de fallos en el sistema, o partes de él, para reparar o reconfigurar la plataforma de manera automática cuando se produzcan errores o fallos internos. Aunque algunos mecanismos de tolerancia a fallos están previstos y especificados en el sistema publicador/suscriptor utilizado para el intercambio distribuido de alertas de nuestra infraestructura (xmlBlaster), no se han podido evaluar durante el transcurso de este proyecto por encontrarse aún en fase de desarrollo. Una vez estén disponibles, se podrá proceder a su configuración y evaluación a través de la realización de nuevos experimentos sobre posibles escenarios de fallo.

Bibliografía

[1] M. Wood, and M. Erlinger. Intrusion Detection Message Exchange Requirements. Internet-Draft. October 2002.

[2] H. Debar, D. Curry, and B. Feinstein. The Intrusion Detection Message Exchange Format. Internet-Draft. March 2006.

[3] A. C. Migus. IDMEF XML library version 0.7.3. *http://sourceforge.net/projects/libidmef*

[4] H. Debar, D. Curry, and B. Feinstein. Intrusion detection message exchange format data model and extensible markup language. Technical report, january 2005.

[5] Y. Vandoorselaere and L. Oudot. Prelude IDS, un Système de Détection d'Intrusion hybride opensource. MISC issue 3, july 2002.

[6] M. Rosech. Snort: lightweight intrusion detection dor networks. In *13th USENIX Systems Administration Conference*, Seatle, WA, 1999.

[7] M. Ruff. XmlBlaster: open source message oriented middleware. White paper [on-line]. *http://www.xmlblaster.org*, 2000.

[8] B. Hammersley. *Content Syndication with RSS*. O'Reilly Ed., First Edition, March 2003 ISBN 0-596-00383-8, 202 pages.

[9] J.Castagnetto, H. Rawat, S. Schumann, C. Scollo, and D. Veliath. *Professional PHP Programming*. Wrox Press Inc., ISBN 1-86100-296-3, 909 pages, 1999.

[10] J. garcía, F. Autrel, J. Borrell, S. Castillo, F. Cuppens, and G. Navarro. Decentralized publish/subscribe system to prevent coordinated attacks via alert correlation. In *Sixth International Conference on Information and*

Communication Security, volume 3269 of *LNCS*, pages 223-235, Málaga, Spain, October 2004. Springer-Verlag.

[11] J. García, M. A. jaeger, G. Mühl, and J. Borrell. Decoupling Components of an Attack Prevention System using Publish/Subscribe. In *2005 IFIP International Conference on Intelligence in Communication Systems*, pages 87-98, Montréal, Canada, Springer-Verlag.

[12] G. Mühl. *Large-Scale Content-Based Publish-Subscribe Systems*. PhD thesis, Technical Univertity of Darmstadt, 2002.

[13] J. Hochberg, K. Jackson, C. Stallins, J.F. McClary, D. DuBois, and J. Ford. NADIR: An automated system for detecting network intrusion and misuse. In *Computer and Security*, volume 12(3), pages 235-248. May 1993.

[14] S. R. Snapp, J. Brentano, G. V. Dias, T. L. Goan, L. T. Heberlein, C. Ho, K. N. Levitt, B. Mukherjee, S. E. Smaha, T. Grance, D. M. Teal, and D. mansur. DIDS (distributed intrusion detection system) - motivation, architecture and an early prototype. In *Proceedings 14th national Security Conference*, pages 167-176, October 1991.

[15] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. GrIDS - a graph-based intrusion detection system for large networks. In *19th National Information Systems Security Conference*, 1996.

[16] G. Vigna and R. A. Kemmerer. NetSTAT: A network-based intrusion detection system. *Journal of Computer Security*, 7(1):37-71, 1999.

[17] G. B. White, E. A. Fisch, and U. W. Pooch. Cooperating security managers: A peer-based intrusion detection system. *IEEE Network*, 7:20-23, february 1999.

[18] R. Shirey. Internet Security Glossary. Informational. May 2000.

Firmado: Ignasi Barrera Caparrós

Bellaterra, Febrero de 2007

Resum

Aquest projecte consisteix en la implementació i avaluació d'una infraestructura de comunicacions per a una plataforma de detecció d'atacs coordinats, basada en el paradigma publicador/subscriptor per a l'intercanvi de missatges IDMEF. Per implementar aquest sistema s'ha fet servir xmlBlaster i s'han desenvolupat les interfícies necessàries per a fer transparent l'accés a la informació de la zarza de comunicacions. El resultat és una plataforma escalable que permet l'intercanvi eficient de informació entre els diferents elements distribuïts del sistema de detecció.

Resumen

Este proyecto consiste en la implementación y evaluación de una infraestructura de comunicaciones para una plataforma de detección de ataques coordinados, basada en el paradigma publicador/suscritor para el intercambio de mensajes IDMEF. Para implementar este sistema se ha utilizado xmlBlaster y se han desarrollado las interfaces necesarias para hacer transparente el acceso a la información de la red de comunicaciones. El resultado es una plataforma escalable que permite el intercambio eficiente de información entre los elementos distribuidos del sistema de detección.

Abstract

This project is about the implementation and evaluation of a communication infrastructure for a coordinated attack detection platform, based on the publish/subscribe paradigm to exchange IDMEF messages. To develop this system we have used xmlBlaster and programmed the necessary interfaces to allow a transparent access to the information in the communication network. The result is an escalable platform that allows an efficient exchange of information between distributed elements of the detection system.