Parte III

Aportación del proyecto

7. Requisitos

A partir de la elaboración de un catálogo de los requisitos del sistema (tanto funcionales como no funcionales), se persigue obtener un modelo conceptual del sistema que describa su funcionalidad y hasta qué límites llegar.

7.1. Catálogo de requisitos

— ***Requisitos funcionales***
Mediante reuniones del equipo de trabajo y el estudio de las necesidades a las que se pretende dar solución con este proyecto, se identifican (referentes únicamente a la Interfaz de usuario) los siguientes requisitos **funcionales**:

RF1 El Sistema debe presentar el estado actual de funcionamiento de cada uno de los Componentes Software que pretende controlar.

RF2 El Sistema debe presentar al Usuario la configuración actual con la que se encuentran funcionando cada uno de los Componentes Software.

RF3 El Sistema debe permitir al Usuario la modificación de los parámetros de configuración que se le ofrezca de cada uno de los Componentes Software.

RF4 El Sistema debe detectar errores en la introducción de los datos de configuración (valores incorrectos, datos incoherentes, etc.) y evitar introducirlos en la configuración de los Componentes Software (a fin de evitar un mal funcionamiento de los mismos). Así mismo, si se da esta situación, debe presentar un mensaje de error al Usuario advirtiendo de este hecho.

RF5 El Sistema permitirá al Usuario el control sobre la *ejecución* de un Componente Software íntimamente relacionado con el Módulo objeto de configuración. En concreto, debe permitir al Usuario el inicio y parada inmediatos de un Componente Software, así como la configuración en el Sistema Operativo de su inicio o parada automáticos (es decir, del arranque del servicio).

— ***Requisitos no funcionales***
Mediante la revisión del entorno tecnológico sobre el que operará la aplicación, y las necesidades y restricciones técnicas con las que se cuenta, se ha elaborado el siguiente catálogo de requisitos **no funcionales**:

Requisitos sobre la arquitectura

RNF1 El sistema debe funcionar sobre una arquitectura Web de 3 capas, y estar desarrollado al completo de manera compatible con el lenguaje PHP y el servidor Web sobre el que éste se ejecuta.

La elección del lenguaje PHP ha sido por motivos de conveniencia (es sencillo de aprender, para utilizar este tipo de tecnologías de 3 capas), ofrece un rendimiento adecuado, y resulta apto para el tamaño del proyecto y la funcionalidad necesaria.

Requisitos sobre la usabilidad

RNF2 El sistema presentará al Usuario un menú o índice que le llevará a las "pantallas" de configuración de cada uno de los Módulos independientes (que controlan a su vez los distintos Componentes Software específicos).

RNF3 Los Módulos emplearán un mecanismo de comunicación entre sí que les permita compartir cierta información (principalmente datos de configuración) con el objetivo de que el Usuario no tenga que introducir un mismo parámetro en múltiples ocasiones (en varias pantallas diferentes).[1]

Este último requisito, tras estudiar lo que supondrá para el desarrollo de la aplicación, nos lleva a determinar un nuevo requisito no funcional (que catalogaremos también como requisito sobre la arquitectura), que resulta de importancia en cuanto a cómo afectará a dicho método de desarrollo:

RNF4 Se empleará Orientación a Objetos en la implementación de los Módulos, con objeto de facilitar la reutilización de código y el *paso de mensajes* entre módulos.

Ello se debe a que la orientación a objetos permitirá independizar con mayor facilidad unos módulos de otros (mediante la propiedad del *encapsulamiento*), unificará interfaces, y permitirá el *paso de mensajes*: así, cuando un módulo requiera que otro adquiera y procese cierta información, se podrá realizar mediante este mecanismo con cierta facilidad.

RNF5 Se empleará XML preferentemente como lenguaje de descripción de estructuras de datos estáticas configurables y/o parametrizables (por ejemplo: para la descripción de los interfaces de usuario, y en los ficheros propios de configuración).

Requisitos sobre el rendimiento
No se definen requisitos sobre el rendimiento para este proyecto.

Requisitos sobre la disponibilidad
No se definen requisitos sobre la disponibilidad para este proyecto. La disponibilidad del sistema la proporcionará principalmente el sistema operativo y el hardware sobre el que se ejecute.

Requisitos sobre la escalabilidad

RNF6 El diseño y la implementación de los Módulos de los que se compone el sistema será suficientemente homogéneo y/o extensible como para permitir la adición de nuevos módulos o nuevas funcionalidades sin que esto suponga modificar gran parte de la aplicación.

Requisitos sobre la seguridad

RNF7 Dado que se trata de un sistema de *Seguridad Perimetral*, se procurará en la medida de lo posible que la implementación se haga de acuerdo a unas prácticas de programación seguras, de forma que se mantenga un nivel mínimo de seguridad en la ejecución del sistema.

[1] De esta forma se evitan posibles errores por inconsistencia en las configuraciones de distintos componentes software. Esta característica es primordial en el Proyecto, y como se hablaba con anterioridad, constituye una importante innovación en este tipo de aplicaciones.

8. Diseño

8.1. Descripción general

Se describe a continuación el proceso seguido y la toma de decisiones realizada en el diseño del sistema, arquitectura e interfaces de usuario a nivel general.

Precondiciones:

Elaboramos un catálogo de precondiciones que habrá que tener en cuenta a la hora de diseñar las interfaces y el sistema, tomando como base el Catálogo de Requisitos del sistema (sección 7.1) y la información con la que contamos hasta el momento en el estudio realizado. Estas precondiciones son:

- Se trata de diseñar interfaces de usuario para la configuración de los diferentes módulos que se integran en el Sistema. Estos son: Proxy, Filtro de contenidos, Cortafuegos, Detector de intrusiones (IDS) y Antivirus de red.
- La interfaz de configuración no abarcará en principio más allá de estos módulos. El resto de los componentes del sistema en todo caso se podrá configurar con los medios ya disponibles en el sistema (SSH, Webmin...) para realizar esta tarea.
- Se pretende que la interfaz de configuración *"integre"* todos los módulos entre sí: un cambio en un módulo que pueda afectar a otro, será notificado a éste último mediante algún mecanismo de comunicación entre módulos.

  Por ejemplo: Al cambiar el puerto de escucha del Proxy, los módulos que se comuniquen con dicha aplicación a través de este puerto (como p.ej. el Filtro de Contenidos) serán notificados de este cambio para que se reconfiguren automáticamente.
- La interfaz de configuración se implementará como una interfaz Web (por simplicidad, versatilidad y conveniencia).
- El lenguaje de implementación de la interfaz Web será *'PHP'*.

  > Resulta adecuado para desarrollar aplicaciones de pequeño/medio tamaño, que además requieren un consumo mínimo de recursos (CPU–Memoria). Es el caso de las aplicaciones *embebidas* (y nuestro sistema puede considerarse como tal).
- Se evitará en la medida de lo posible el empleo de un motor de bases de datos en el sistema final en ejecución.

  > Igualmente, no resulta adecuado para aplicaciones embebidas, y no es realmente necesario para el alcance inicial del proyecto, desde el punto de vista del análisis.
- Se intentará separar la presentación del control (o *lógica de negocio*) mediante *plantillas* que independicen el diseño de la interfaz de la aplicación en sí, así como descripciones en lenguaje XML para especificar estructura en lugar de presentación.

Con esta información en la mano, podemos establecer unos límites claros de hasta dónde abordará el trabajo en las fases siguientes, y estos datos de nivel arquitectónico resultarán así mismo extremadamente clarificadores para resolver muchos problemas.

8.2. Selección de módulos

Componentes software estándar

Los componentes software elegidos para la realización de la funcionalidad de Seguridad requerida, en cada uno de los casos propuestos, son:

- **Cortafuegos:** iptables / FireHOL
- **Proxy:** Squid
- **Filtro de contenidos:** Dansguardian
- **Detector de intrusiones:** Snort
- **Antivirus:** ClamAV

Dichos componentes Software se han elegido de los existentes en el mundo *Open-Source* por su calidad y conveniencia para el proyecto, ratificando su idoneidad en cuanto a llevar a cabo las tareas de seguridad que se piden en cada caso. Para ello, hemos realizado un estudio exhaustivo de cada uno, con objeto de conocer sus necesidades de configuración (lo cual que será indispensable conocer para la construcción de la Interfaz cuya misión principal es facilitar dicha configuración), y que presentaremos en las siguientes secciones. Se trata de las aplicaciones de código abierto más maduras (excepto FireHOL) y completas en su campo, llegándose a utilizar extensivamente en multitud de proyectos que buscan esta funcionalidad, lo cual es ya de por sí una garantía. Esta última razón, de suficiente peso para tener en cuenta, es la que nos ha impulsado a considerar el contar con estas aplicaciones en primera instancia.

El caso de FireHOL es algo distinto: se trataba de buscar una forma de configurar el cortafuegos de Linux que resulte sencilla, independiente de la plataforma en la medida de lo posible, a la vez que flexible. `iptables` (la implementación actual de `netfilter` de Linux) no reunía estas características, y pensamos que sería más apropiado utilizar una interfaz de configuración que abstrajera las interioridades de la gestión del cortafuegos al más alto nivel. Después de evaluar varias alternativas, consideramos FireHOL como la más atractiva a la vez que novedosa, siendo capaz de aprovechar toda la potencia del cortafuegos de Linux, con una sintaxis sencilla y flexible (luego lo veremos en más detalle).

A continuación veremos el estudio realizado sobre cada uno de los módulos.

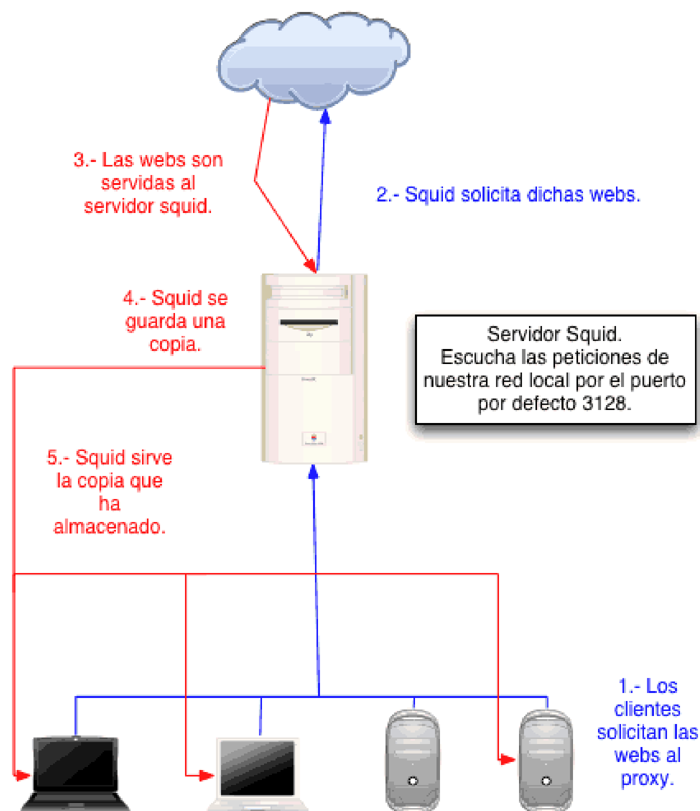

Figura 3: Funcionamiento Squid

8.3. Descripción del módulo Squid

8.3.1. Descripción de Squid

Squid es un software que almacena ("cachéa") datos de Internet. Lo hace guardando las peticiones que los usuarios realizan. En otras palabras, si una persona quiere descargar una página web, pide a Squid que obtenga dicha página. Squid conecta con el servidor remoto (por ejemplo http://www.barrapunto.com/) y pide la página. Después reenvía la petición al usuario, pero al mismo tiempo mantiene una copia. La próxima vez que alguien desee dicha página, Squid simplemente la lee del disco y la transfiere al usuario de forma instantánea.

Squid soporta actualmente los protocolos HTTP, FTP, GOPHER, SSL y WHAIS. Y no soporta otros protocolos como RealAudio, Streamings y similares.

En el diagrama de la figura 3 vemos resumido el funcionamiento del proxy.

El proceso de instalación del proxy Squid en un sistema Debian (o basado en Debian como es nuestro caso), es simple si hacemos uso del gestor de paquetes *apt* (u otros como *aptitude*).

Durante el desarrollo de este proyecto siempre se trato de usar software precompilado en paquetes ".deb". Incluso en algunos casos empaquetamos software no encontrado en este formato. El motivo es que así se facilita mucho la gestión de paquetes: instalación, actualización y eliminación de los mismos.

En el anexo C.1 en la página 91 de esta memoria, se describe con mayor detalle el proceso de instalación de Squid, así como el proceso de creación o reconstrucción de su caché. También se explica en este punto como configurar el sistema para arrancar el proxy Squid al inicio.

La configuración de Squid se encuentra en el archivo *squid.conf*, en nuestro caso está en */etc/squid/squid.conf*. Tras las instalación encontramos un fichero extenso con multitud de opciones comentadas.

En el anexo C.2 en la página 91 de esta memoria, aclaramos las principales agrupándolas según su funcionalidad. Podemos consultar un manual más extenso y detallado sobre la configuración de Squid en: Manual de configuracion de squid `http://squid.visolve.com/squid/squid24s1/contents.htm`.

Después de ver todos los parámetros de configuración de Squid, nos vemos obligados a seleccionar los más importantes. Los demás tomarán el valor por omisión. De los parámetros elegidos algunos se dejarán a un valor fijo en el fichero de configuración y otros serán configurables desde la aplicación web de configuración.

Esta lista de parámetros principales puede encontrarse en el anexo C.3 en la página 97. En el se enumeran los principales parámetros del fichero squid.conf, sus implicaciones en el funcionamiento del proxy y sus valores por omisión (*default values*).

8.3.2. Reflexión sobre funcionalidades y parámetros

Llegados a este punto, tenemos un amplio conocimiento de las potenciales funcionalidades que ofrece Squid. Este conocimiento obtenido a partir del estudio exhaustivo de sus parámetros de configuración nos permite seleccionar que funcionalidades serán configurables desde la interfaz de nuestra aplicación web.

A continuación veremos algunas de estas funcionalidades:

- **Funcionando como proxy transparente:**

En un proxy normal los clientes deben especificar el host y el puerto del proxy en su navegador, de manera que los navegadores web realicen sus peticiones al proxy y no a los servidores web. Sin embargo, esto puede no ser lo que queremos en algunas situaciones:

- Puede ser que queramos que los usuarios del proxy lo usen quieran o no quieran.
- Que no sean conscientes de estar usándolo.
- O simplemente se quiere evitar el trabajo que supone configurar todos los navegadores de los clientes.

Para solucionar el problema anterior podemos usar el proxy en modo transparente, de esta manera los clientes no serán conscientes de estar usando el proxy, bastará con que usen como puerta de enlace predeterminada el host donde está nuestro proxy. Squid interceptará las peticiones de forma transparente.

Hemos de saber que el proceso es transparente para los clientes, ya que el servidor sabe que está hablando con el servidor proxy, y verá la dirección IP del proxy no la de los clientes. Aunque Squid puede pasar una cabecera *X-Forwarded-For*, de forma que el servidor conocerá la IP del usuario del proxy.

Para que nuestro servidor Squid actúe como proxy transparente debemos redirigir las conexiones TCP a puertos locales, pero además es necesario que avisemos al servidor proxy, Squid, que estamos funcionando en modo transparente para que pueda establecer conexiones con los servidores originales.

Por último hemos de decir que no se podrá cachear de forma transparente peticiones HTTPS (páginas seguras que utilicen los protocolos SSL, TSL, etc.). Esto es debido a problemas de seguridad relacionados con ataques del tipo "man-in-the-middle".

Configuración del Firewall:

La regla de iptables que habría que añadir es la siguiente:

iptables -t nat -A PREROUTING -i eth0 -p tcp –dport 80 -j REDIRECT –to-port 3128

Nosotros usaremos fireHOL como capa de abstracción de iptables, para más información sobre fireHOL remitimos al lector a la sección.

Un comando de fireHOL que nos puede resultar muy útil a la hora de hacer funcionar Squid como proxy transparente es *transparent_squid*.

Por lo tanto hemos de añadir la siguiente linea en el archivo de configuración de fireHol:

```
transparent_squid 8080 "proxy"
```

donde 8080 es el puerto en que escucha Squid, por defecto sería 3218, pero en este caso hemos supuesto que DansGuardian, el filtro de contenidos, está corriendo delante de Squid escuchando en su puerto por defecto 8080.

- **Usando antivirus para filtrar el tráfico:**

Una posibilidad que contemplamos es el filtrado del tráfico que pasa a través del proxy con un antivirus. El antivirus puede ser externo o puede estar corriendo en nuestra máquina. Más adelante profundizaremos en este tema, y las opciones a añadir en el archivo de configuración de Squid pueden verse en la sección E.2 en la página 111.

- **Parche para X-forwarded-for:**

Al paquete original de Squid se le ha aplicado un parche para activar el reconocimiento de las cabeceras X-Forwarded-For para que tengan efecto en lo expresado en las ACLs. Esto ha sido necesario para permitir la integración correcta con DansGuardian. Para más información sobre el problema encontrado y la solución adoptada véase la sección 8.4.1.

- **Métodos de autenticación:**

Nosotros en principio no contemplaremos la posibilidad de que los usuarios se autentiquen para acceder al servidor Squid, no obstante comentaremos someramente esta posibilidad, que puede tenerse en cuenta en futuras ampliaciones y mejoras del proyecto.

La configuración por defecto de Squid permite que cualquier usuario tenga acceso sin necesidad de realizar proceso alguno de autenticación. Para autenticar a los usuarios, (por ejemplo para que solo un grupo de usuarios, desde cualquier máquina en la red, pueda navegar por Internet), Squid permite autenticación con nombre de usuario y contraseña pero a través de una aplicación externa, usando la ACL *proxy_auth* y *authenticate_program*; se fuerza a un cliente a verificar nombre de usuario y contraseña antes de que obtenga acceso a Internet. Hay varios programas de autenticación disponibles que Squid puede usar :

- LDAP : Usa Linux Lightweight Directory Access Protocol

- NCSA : Usa un archivo estilo NCSA con username y password
- SMB : Usa el server SMB server como SAMBA Windows NT
- MSNT : Usa la autenticación de dominio de Windows NT
- PAM : Usa Linux Pluggable Authentication Modules
- getpwam : Usa el archivo de contraseñas de Linux

Se necesita especificar el programa de autenticación que será usado especificando la opción *authenticate_program*. El programa que se va a usar para la autenticación debe estar instalado y funcionando. A continuación vemos parte del archivo de configuración para un caso particular:

> *authenticate_program /usr/local/bin/pam_auth*
> *acl pass proxy_auth REQUIRED*
> *acl mynetwork src 192.168.0.1/255.255.255.0*
> *http_access deny !mynetwork*
> *http_access allow pass*
> *http_access deny all*

El anterior ejemplo usa el programa de autenticación PAM y todos los usuarios necesitan autenticarse antes de salir a Internet.

Opciones como *authenticate_ttl* y *authenticate_ip_ttl* también pueden ser usadas para cambiar el comportamiento del proceso de autenticación, por ejemplo revalidar el usuario y su contraseña.

8.3.3. Clasificación de parámetros

Finalmente comentamos qué parámetros serán configurables directamente por el usuario y cuáles no y qué parámetros tendrán un valor fijo o “automático”. Además especificaremos hasta qué punto puede el usuario configurar un parámetro dado.

Puerto de escucha: será un parámetro configurable por el usuario desde la interfaz de configuración. Se controla con la directiva *http_port*, que solo recibe un parámetro, el número de puerto.

Tamaño de la caché en RAM: Es otro parámetro configurable desde la interfaz, se controla por medio de la directiva *cache_mem*, que solo recibe un parámetro y es precisamente el tamaño de la caché en RAM.

Directorio de caché: el lugar que ocupa la estructura de directorios de caché será fijo, */var/spool/squid*, y el usuario podrá variarlo editando el archivo de configuración *squid.conf*, no desde la interfaz de configuración. Se fija mediante un parámetro en la directiva *cache_dir*, por ejemplo: *cache_dir ufs /var/spool/squid 100 16 256*

Sistema de almacenamiento: el sistema de almacenamiento también fijado por defecto como *ufs*, se controla con la misma directiva que en el caso anterior.

Tamaño de caché en disco: la directiva que lo fija es la misma que en los anteriores casos, *cache_dir*, en este caso será accesible desde la interfaz de configuración para ser modificada por el usuario.

Tamaño máximo de objeto cacheable: se da oportunidad de configurarlo desde la interfaz, la cual tendrá que modificar la directiva *maximum_object_size.*

Proxy transparente: el usuario puede elegir este modo de funcionamiento desde la interfaz de configuración, si es así habrá que añadir las siguientes lineas:

```
httpd_accel_host virtual
httpd_accel_port 80
httpd_accel_with_proxy on
httpd_accel_uses_host_header on
```

Antivirus: si se decide examinar el tráfico en busca de virus, lo cual es posible desde la interfaz de configuración del antivirus, se deben incluir las siguientes lineas en *squid.conf*, (obsérvese que se fija el número de procesos de redirector que correrán, así como la ubicación del script redirector y su fichero de configuración):

```
redirect_program /usr/local/bin/SquidClamAV_Redirector.py -c /etc/squid/SCAVR.conf
redirect_children 25
```

Follow_x_forwarded_for: se incluirán las siguientes lineas fijas por defecto, con la intención de que Squid interprete las cabeceras *x_forwarded_for*:

```
follow_x_forwarded_for allow localhost
follow_x_forwarded_for allow miredlocal
acl_uses_indirect_client on
delay_pool_uses_indirect_client on
log_uses_indirect_client on
```

Proxy padre: se da la oportunidad de configurar el uso de un proxy caché padre, del que solo se puede especificar la dirección y el puerto a través de la interfaz de configuración. En el archivo de configuración tendremos que modificar la directiva *cache_peer*. Y se añadirá también la directiva *never_ direct allow all*, esto hace que Squid nunca mande directamente la petición a la fuente, sino que trate de buscar un proxy padre al que enviar esta petición, si no encontrase proxy padre devolverá un error.

Puertos permitidos: se permite al usuario definir los puertos navegables desde la interfaz de configuración, las lineas generadas son una acl y las http_access relacionadas, se muestra a continuación:

```
acl puertos_permitidos port 80 81 119 210 443 563 3389 1024-65535
http_access deny !puertos_permitidos
```

Reglas de navegación: se permite crear, borrar, editar y mover (el orden de aparición de las reglas en el archivo es importante), de reglas de navegación. Esto involucra dos tipos de directivas *acl* y *http_ access*. Al menos es necesario la existencia de una acl que englobe todo, en nuestro archivo será por defecto: *acl all src 0.0.0.0/0.0.0.0*.
Cada regla tiene un número de orden, ya que éste es importante a la hora de establecer prioridades o excepciones, este número fija la posición de la directiva http_access de esta regla respecto a las demás.
Se pueden indicar direcciones IP de origen (o direcciones de red con máscara de bits), y sitios de destino (es decir, nombres de dominio), esto se refleja en la directiva acl donde aparecerán tras los parámetros *src* o *dst*.
Por último, puede tratarse de una regla que permita o deniega el acceso, esto se indica en el primer parámetro de la directiva http_access que se fijará a *allow* o *deny* según proceda.

Regla de navegación por defecto: Es la *última regla de navegación*, que se aplica una vez que se ha recorrido la lista al completo y no se ha encontrado una concordancia. Si el usuario prefiere que se deniegue por defecto la navegación excepto para aquellos puestos o destinos indicados expresamente, entonces pondrá aquí *"Denegar"* en este caso se añadirá al final de todas las reglas, *http_ access deny all* . Si por el contrario, prefiere permitir la navegación como regla general excepto en aquellos casos que se deniegue expresamente, entonces pondrá aquí *"Permitir"* y la última regla de todas será http_access allow all.

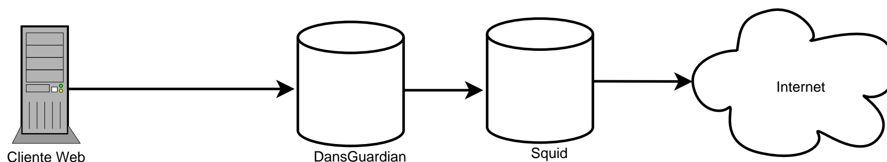

Figura 4: Integración de DansGuardian + Squid

8.4. Descripción del módulo DansGuardian

8.4.1. Descripción de DansGuardian

Los filtros de contenido resuelven los problemas de seguridad de la información, analizan los datos que pasan a través de ellos y deciden que hacer, dejarlos pasar, bloquearlos o llevar a cabo otras posibles acciones.

No solo interpretan las cabeceras de los paquetes, URLs o IPs, etc, sino que inspeccionan el contenido de los mismos, es ahí donde reside su potencia y utilidad.

DanGuardian es un filtro de contenido de páginas web, que no filtra simplemente un grupo de sitios restringidos. Está basado en varios métodos de filtrado como coincidencia de palabras, filtrado por PICS (Plataforma para la Selección de Contenido de Internet) o por URL.

DG está diseñado para ser flexible a la hora de ajustar el filtrado a nuestras necesidades, que puede ser absolutamente "draconiano" o prácticamente imperceptible. La configuración por defecto está pensada para las necesidades de una escuela de primaria, pero como hemos comentado pueden variarse y ajustarse a cualquier necesidad.

DansGuardian se sitúa entre el cliente web y el proxy interceptando y modificando su comunicación. Squid escucha por defecto en el puerto 3128 y DG escucha en el 8080 por defecto. Cuando DG recibe una petición en el puerto 8080, filtra dicha petición y la pasa a Squid por el puerto 3128, (estos puertos son configurables tanto en Squid como en DG, para esta introducción usamos los valores por defecto).

La intercepción de la comunicación puede realizarse simplemente configurando los clientes para que hagan las peticiones al puerto 8080 o bien habilitado el filtrado transparente en Squid y redirigiendo el tráfico saliente en el firewall del puerto 80 al 8080 de localhost, (presuponiendo que DG está en el firewall). El puerto 443 necesitara también ser redireccionado al igual que otros puertos de proxy comunes deberán ser bloqueados para prevenir que se traspase el filtrado.

En la figura de la página 37 podemos ver un esquema que aclara el flujo de datos desde el cliente hasta el servidor web, atravesando el filtro de contenidos y el proxy.

Interesa que DansGuardian esté situado entre el cliente Web y el Proxy ya que, en caso contrario, podríamos encontrarnos con problemas de seguridad en el acceso a la información por parte de los usuarios. Se podría dar el caso de que, si a un usuario se le ha permitido el acceso a una página Web, esta quedase almacenada en la caché del Proxy. El siguiente usuario que intente acceder a dicha página, la obtendría directamente de la caché del Proxy, sin llegar a acceder al filtro de contenidos, a pesar de que para éste usuario puedan existir reglas de filtrado diferentes que bloqueasen su visualización. Esta es la razón principal de que DansGuardian deba estar situado antes de cualquier Proxy-caché.

DansGuardian se distribuye bajo licencia GPL, de forma gratuita para usos no comerciales y de forma no gratuita para usos comerciales. Podemos encontrar más información sobre DansGuardian en `http://dansguardian.org/`.

La instalación del software DansGuardian, al igual que Squid, se realizo usando la herramienta apt. Pueden consultarse más detalles sobre la instalación de Dansguardian en el anexo D.1 en la página 101 de esta memoria.

Antes de instalar DG es conveniente que Squid esté instalado en el sistema, además la configuración de Squid y Dansguardian estará interrelacionada.

También será necesario un servidor httpd que ejecute un script cgi escrito en perl, este notificará al usuario en caso de haber un acceso denegado mostrándole una página web de información.

Un inconveniente de integrar DansGuardian y Squid, es que DG anula las ACL's (listas de control de acceso) de Squid. Trataremos de explicar el caso con más detalle: DG se coloca entre los clientes y el proxy Squid, de manera que a Squid todas las peticiones le llegarán con la IP del host donde corre DG, en nuestro caso *localhost,* esto impide que Squid controle el acceso por IPs a los clientes ya que se ha perdido esta información. Por otro lado los archivos de log de Squid también se verán falseados por este motivo.

La solución que decidimos adoptar para evitar los problemas anteriores se basa en la cabecera de http, *X-Forwarded-For*, que contiene la IP del cliente que realizó la petición. Sin embargo Squid no es capaz de interpretar por defecto esta cabecera, por lo que es necesario parchearlo. El parche de Squid que añade la anterior funcionalidad está disponible en `http://squid.sourceforge.net/follow_xff/`.

Los archivos de configuración y listas de expresiones, IPs, sitios, etc, de DansGuardian están ubicados en el directorio */etc/dansguardian/.* En el anexo D.1 en la página 101 se puede encontrar un listado de los archivos de configuración que podemos encontrar en este directorio.

El archivo de configuración principal es *dansguardian.conf*, sin embargo hay otros que también nos resultaran de interés, se detallan estos archivos y su contenido en el anexo D.2 en la página 101 de esta memoria.

También en el anexo anteriormente citado se puede encontrar una descripción de otros ficheros de configuración, mostrándose especial detalle en aquellos que contienen listas (de urls, sitios, direcciones IP's, de expresiones, etc).

En el caso de las listas de expresiones (bloqueadas, permitidas y ponderadas) se explica la sintaxis en la que dichas expresiones deben ser introducidas. Según se introduzca cada expresión el comportamiento del filtro puede variar notablemente. Por ejemplo:

< test>cualquier palabra que empiece por 'test'
<test >cualquier palabra que acabe en 'test'
<test>cualquier palaba que contenga 'test'
< test >solo la palabra 'test'
<this is a test phrase>la frase exacta
<test>,<secondtest>siempre que se encuentren las dos palabras en la pagina.

8.4.2. Reflexión sobre los parámetros y clasificación de estos.

Comenzaremos comentando lo referente a las directivas del archivo *dansguardian.conf.* Todas las directivas que aparecerán en este archivo se pueden ver en la sección D.2 en la página 101. La mayoria de dichas directivas se fijarán a su valor por defecto, este es el que aparece en el anexo anteriormente citado

A continuación comentaremos aquellos que se fijarán a otro valor diferente del valor por omisión, o aquellos que podrán ser modificados desde la interfaz de configuración. Finalmente en esta misma sección, se comentarán de igual forma las directivas principales de los demás archivos de configuración.

Puerto del filtro DG: desde la interfaz de configuración se podrá configurar el puerto en el que escuchará DG, por defecto 8080, la directiva a modificar es *filterport.*

Datos del proxy: DG necesita un proxy al que realizar las peticiones de las páginas que atraviesen el filtro, este proxy puede ser local o estar en otra máquina. Los datos, IP y puerto, se configuran desde la interfaz modificando las directivas: *proxyip* y *proxyport*.

Xforwardedfor: es necesario incluir *X-Forwarded-For: <clientip>* en las cabeceras http, así como interpretarlas por los motivos ya comentados en la sección 8.4.1 en la página anterior, por tanto en el archivo dansguardian.conf estarán las siguientes lineas:

forwardedfor = on
usexforwardedfor = on

Direcciones IP's para saltarse el filtro: si tenemos distintas URL's bloqueadas por nombre, los clientes pueden en principio saltarse el filtro poniendo la ip en las barras de direcciones de sus navegadores en lugar de la URL; es decir si tenemos *www.google.com* bloqueada, los usuarios pueden acceder a esta pagina con solo escribir *http://216.239.59.104/*.
Para evitar esta situación indeseable, una posible solución es bloquear todas las peticiones que contengan la dirección IP en lugar de una URL, esto se consigue añadiendo la linea: **ip,* en el archivo */etc/dansguardian/bannedsitelist*.
Otra solución menos restrictiva es activar la resolución inversa de DNS, se comento la funcionalidad de la directiva en cuestion en D.2 en la página 104, esta opción sin embargo puede reducir la velocidad de busqueda y se desaconseja su uso si no se tiene un servidor de nombres local. Nosotros optaremos por la segunda opción por defecto, sin embargo se da la posibilidad en la interfaz de configuración de tomar también la primera opción impidiendo el acceso a sitios mediante la IP númerica, que por restrictiva pude causar ciertos inconvenientes a los usuarios pero puede resultar ser la opción más segura y eficaz en algunas circunstancias.

Grupos especiales de usuarios: para simplificar, habrá 2 grupos especiales de usuarios cuyas peticiones se procesarán de forma especial. Los usuarios con "acceso sin filtro", que atravesarán el filtro sin ninguna restricción, y sus IP's (no sus nombres de host), se recogerán de la interfaz para ser incluidas en el archivo: */etc/dansguardian/exceptioniplist*.
Otro grupo especial serán aquellos usuarios cuyo acceso será restringido por completo, son los usuarios "sin acceso" o usuarios *baneados*. Sus IP's también se podrán introducir en la interfaz de configuración para incluirlas en el archivo */etc/dansguardian/bannediplist*.

Filtros de URL's: se permite a los usuarios definir mediante la interfaz distintos sitios web y URL's que se bloquearán siempre independientemente de su contenido. Estas se deben incluir en los archivos */etc/dansguardian/bannedsitelist* si la cadena introducida es un *"sitio web"* y en */etc/dansguardian/bannedurllist* si son una URL. Se distingue si es un sitio o un URL según se encuentre el caracter "/" en la cadena, en cuyo caso se considera una URL. No se deben introducir en el archivo de configuración los indicativos de los protocolos http:// o www.
Así mismo, se definirán a través de la interfaz, sitios y URL's que se serviran siempre independientemente de su contenido. Estos se incluirán con criterios similares a los anteriores en los archivos: */etc/dansguardian/exceptionsitelist* y */etc/dansguardian/exceptionurllist*.

Navegación por URL's con direcciones IP: como ya comentamos anteriormente en 8.4.2, se da la opción de impedir este tipo de navegación.

Modo de lista blanca: Es posible dejar unos sitios abiertos, los indicados en los archivos */etc/-dansguardian/exceptionsitelist* (o también en */etc/dansguardian/greysitelist*), y bloquear los demás por defecto. Este modo de funcionamiento se conoce como modo *"blanket block"* o lista blanca, se consigue añadiendo la linea: "**", en el archivo *bannedsitelist* (lo cual significa que se bloquearán todos los sitios por defecto).

Filtros de contenidos: El usuario va a poder definir en la interfaz de configuración las frases prohibidas, que de ser encontradas en una página causaran su bloqueo, estas debemos introducirlas en el archivo */etc/dansguardian/bannedphraselist.* El formato a seguir ya se ha comentado anteriormente en la sección D.2 en la página 105 (véase la descripción de este fichero).
De igual forma se podrá especificar un conjunto de palabras o expresiones *"privilegiadas"*, que de ser encontradas en una página causan que esta atraviese el filtro sin más. Estas palabras las introducimos en el archivo */etc/dansguardian/exceptionphraselist.*
Por último se permite en la interfaz definir frases o palabras ponderadas con un valor, estas irán al archivo */etc/dansguardian/weightedphraselist.*

Límite máximo en la ponderación: si la suma de los valores asociados a las expresiones ponderadas encontradas en una página supera un cierto límite la página no se servirá al cliente, este límite lo podrá indicar el usuario en la interfaz y se corresponde con la directiva *naughtynesslimit*, que podemos encontrar en el archivo */etc/dansguardian/dansguardianf1.conf.*

8.5. Descripción del Módulo Antivirus (ClamAV)

8.5.1. Introducción

Los antivirus son programas cuya función es detectar y eliminar virus informáticos y otros programas maliciosos (a veces denominado malware).

Básicamente, un antivirus compara el código de cada archivo con una base de datos de los códigos de los virus conocidos, por lo que es importante actualizarla periódicamente a fin de evitar que un virus nuevo no sea detectado.

También se les ha agregado funciones avanzadas, como la búsqueda de comportamientos típicos de virus (técnica conocida como heurística) o la verificación contra virus en redes de computadoras.

Los antivirus son esenciales en sistemas operativos cuya seguridad es baja, como Microsoft Windows, pero existen situaciones en las que es necesario instalarlos en sistemas más seguros (tipo Unix, GNU/Linux y similares).

En nuestro caso el antivirus a usar es ***ClamAV*** `http://www.clamav.net/`, también correrá en nuestro sistema el demonio ***freshclam*** que se encarga de mantener actualizada la base de datos de virus.

Y por último será necesario usar un *redirector* que integre ClamAV con Squid, de manera que llame al antivirus cada vez que el proxy lo necesite. Usaremos ***SquidClamavRedirector*** *`http://www.jackal-net.at/tiki-read_article.php?articleId=1`*, este redirector hace uso de ***pyClamAV***, `http://xael.org/norman/python/pyclamav/index.html`, como *binding* a la biblioteca *libclamav*.

8.5.2. ClamAV

Clam Antivirus es una herramienta para sistemas UNIX, distribuida bajo licencia GPL. Proporciona un demonio multihilo escalable y un escáner en linea de comandos. Además de esto, incorpora una avanzada herramienta para la actualización automática desde Internet, "freshclam".

ClamAV detecta más de 25000 virus, gusanos y troyanos, incluyendo virus de macro MSOficce y MacOffice. También escanea archivos comprimidos: zip, rar, tar, gzip, bzip.

ClamAV soporta *"onaccess-scanning"*, con solo cargar un modulo extra en nuestro núcleo, dazuko.o. Podemos descargar las fuentes de este modulo de `http://www.dazuko.org`, dazuko es un driver GPL que permite el control de acceso a ficheros y aplicaciones desde userland. Sin embargo en nuestro caso no necesitaremos esta funcionalidad, ya que será el redirector el que llame a Clamav cuando el Proxy (Squid) lo requiera.

8.5.3. Base de datos de virus. FreshClam.

La base de datos de virus de clamav, es mantenida por el equipo *virusdb* de clamav. Siempre que alguien encuentre un virus que no es detectado por clamav, debe comunicarlo rellenando el formulario que puede encontrar en: `http://www.clamav.net/sendvirus.html`.

Las actualizaciones de la base de datos de virus son frecuentes, no menos de tres por semana. Si se quiere estar enterado de cuando se producen las actualizaciones podemos suscribirnos a la lista clamav-virusdb en lists.clamav.net.

En un ejemplo del log de una actualización realizada en la fecha de redacción de este documento, podemos comprobar que la base de datos tiene aproximadamente 30000 virus registrados:

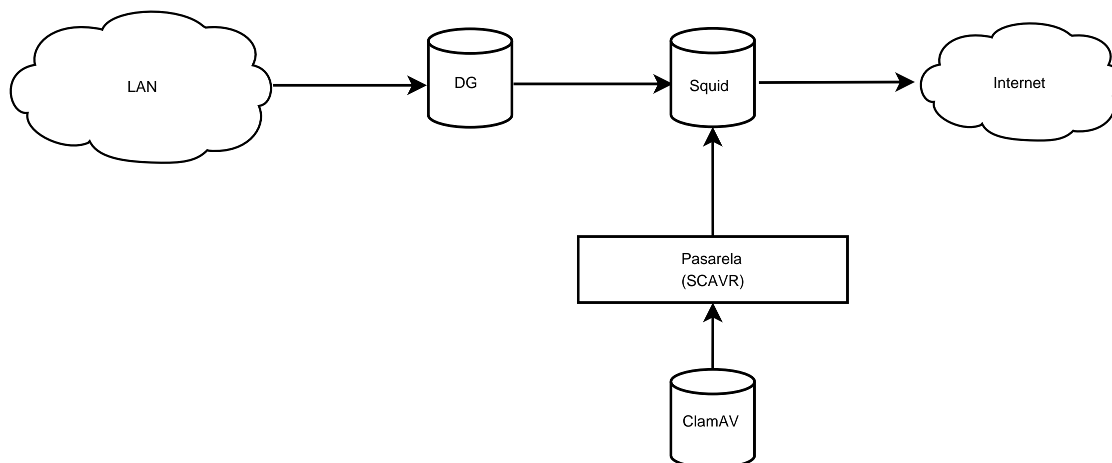

Figura 5: Integración de ClamAV en el flujo de datos.

———————————————

freshclam daemon 0.80 (OS: linux-gnu, ARCH: i386, CPU: i386)

ClamAV update process started at Sun Jan 23 20:57:28 2006

Received signal 15, terminating

Received signal 15, terminating

Received signal 15, terminating

Received signal 15, terminating

main.cvd updated (version: 29, sigs: 29086, f-level: 3, builder: tomek)

daily.cvd updated (version: 679, sigs: 506, f-level: 3, builder: tkojm)

Database updated (29592 signatures) from db.local.clamav.net (213.203.254.4).

———————————————

8.5.4. Squid ClamAV Redirector. SCAVR

La intención inicial es hacer que el tráfico web además de ser filtrado con DansGuardian, sea escaneado en busca de posibles virus, al menos ciertos archivos descargados susceptibles de estar infectados.

Planteamos dos posibilidades, o bien usar clamav como antivirus, configurando Squid para que así sea. O bien usar un antivirus externo que incluso puede estar en otro host. En el primer caso es necesario integrar el antivirus en el flujo de datos desde Internet hacia nuestra red, esta cuestión la trataremos a continuación.

Decidimos integrar el antivirus (clamav) con Squid y no con DG ya que Squid es el elemento más exterior en la secuencia de procesado del tráfico.

La idea se refleja en el gráfico en esta página.

De manera que necesitamos una pasarela que haga que ClamAV escanee los archivos necesarios, las opciones que tenemos son:

- Programar nosotros mismos una pasarela: no porque complicaría la solución.

- Viralator: es un redirector. Algunas cosas como que usa para descargar wget y recurre a popups para mostrar información, no nos dieron buena impresión.
- SafeSquid: es un filtro de contenidos en si mismo, deberíamos cambiar DG por este y supondría replantearlo todo el diseño de nuestro sistema.
- DansGuardian Antivirus Patch: otra posibilidad es integrar clamav con DG, para esto necesitamos parchear DG. El problema es que al estar detrás de Squid, la cache de este no estaría libre de virus.
- SquidClamAv Redirector: Esta es la opción elegida, de todas las posibles para integrar con Squid es la más simple y efectiva.

SquidClamAV Redirector es un script de ayuda a Squid que le añade la posibilidad de escanear determinados ficheros en busca de virus. Squid debe ser configurado para usar programas externos, (opción redirect_program), y también para impedir el acceso a las URL definidas en el archivo de configuración de SCAVR o direccionar las peticiones a una página que muestra el error y diversa información. SCAVR maneja las peticiones dadas por Squid, descarga la URL y la escanea en busca de virus conocidos, en caso de encontrar virus desvía la petición a la página de información.

Referencia de SCAVR: `http://www.jackal-net.at/tiki-read_article.php?articleId=1`

Una vez descritos todos los componentes del módulo antivirus (antivirus, librerias, redirector y demonio de actualización), procedemos a describir el proceso de instalación y configuración de cada módulo.

Con el fin de no romper con el ritmo narrativo de esta memoria, estos detalles técnicos se han sacado al anexo E.1 en la página 107 de esta memoria. En este anexo se describe el proceso de instalación de cada uno de los módulos y los detalles fundamentales de configuración.

Los principales archivos de configuración son:

- /etc/clamav/freshclam.conf. (Configuración del demonio de actualizaciones).
- /etc/squid/SCAVR.conf (Configuración del redirector scavr).

También será preciso añadir algunas lineas en el archivo de configuración del proxy Squid. Esto ya se comentó en la sección de este módulo.

En el anexo E.2 en la página 109, podemos encontrar descritos en detalle los parámetros que encontraremos en los archivos de configuración de los componentes de este módulo antivirus.

8.5.5. Reflexión sobre los parámetros y clasificación de estos

A continuación comentamos como se relacionan los párametros unos con otros, cuáles son importantes para lograr la funcionalidad deseada o requerida, etc. Así mismo documentamos cuales serán configurables directamente por el usuario y cuáles no. Qué parámetros tendrán un valor fijo o "automático", y sus valores posibles, es decir, hasta qué punto puede el usuario configurar un parámetro dado.

Los archivos de configuración de las directivas que se ven afectadas por este módulo de la interfaz son */etc/clamav/freshclam.conf* y */etc/squid/SCAVR.conf*. Ejemplos de estos archivos podemos verlos en E.2 en la página 110. Todo párameto que no sea comentado en esta sección se considerará fijado al valor por defecto que aparece en las secciones anteriormente referenciadas.

Filtrar contra virus el tráfico web: el antivirus ClamAV no es un servicio que debamos arrancar, por lo tanto cuando el usuario decida activar el filtrado en la interfaz de configuración, lo que haremos es incluir en la configuración de Squid, el archivo */etc/squid/squid.conf*, las directivas necesarias para que se enganche al antivirus por medio del redirector. Esto se explica en la sección E.2 en la página 111.
Los párametros de la primera directiva introducida serán fijos ya que la ubicación del script y del archivo de configuración está fijada. En la segunda no dejaremos libertad al usuario para configurar los párametros desde la interfaz; *redirect_ children* indica el número de procesos del programa de redireccion que Squid arrancará, si es demasiado alto consumira demasiada RAM pero si es muy bajo Squid puede perder mucho tiempo esperando a que se precesen los archivos en busca de virus. El valor por defecto al que fijamos esta directiva será 45, pero el valor optimo dependera de la carga del servidor y la memoria de este, por lo tanto el usuario avanzado editará el archivo de configuración para adaptar este valor a las necesidades del sistema.

Actualizaciones: Solo se dará oportunidad al usuario de configurar las actualizaciones de dos de las posibles formas de hacerlo, (ver la sección E.2 en la página 109, para más información sobre las posibilidades), estas son correr freshclam como demonio (lo más recomendable), o actualizar manualmente la base de datos. En el primer caso deberemos arrancar el demonio con "*/etc/init.d/clamav-freshclam start*".
En caso de que el usuario eliga la opción de correr el demonio, se le da también la posibilidad de introducir el número de actualizaciones diarias, esto es la directiva *Checks* del archivo

/etc/clamav/freshclam.conf.
También es configurable la dirección del servidor al que conectarse para obtener la actualización, esta dirección queda reflejada en la directiva *DatabaseMirror, por defecto: DatabaseMirror* db.local.clamav.net. Podemos mostrarle al usuario distintas urls de bases de datos obteniendolas de */var/lib/dpkg/info/clamav-freshclam.templates.*
Hemos de tener en cuenta para la configuración del Firewall, que freshclam se conectara al puerto 80 del repositorio elegido.
Por último, se da la oportunidad de que el usuario decida si quiere que freshclam se conecte a internet en busca de actualizaciones directamente o por medio del proxy padre definido en la configuración de Squid. De elegirse la opción de usar el proxy padre, ya sea porque es la única via de salir hacia el exterior desde la red donde nos encontremos o por otro motivo, hemos de incluir las directivas siguientes en freshclam.conf:

HTTPProxyServer proxyhost
HTTPProxyPort proxyport

Tamaño máximo de los archivos: Otro parámetro configurable será *MaxRequestsize del archivo* /etc/squid*/etc/squid/SCAVR.conf,* que fija el tamaño máximo de los archivos precesados en busca de virus por Clamav.

Extensiones de archivos escaneables: También dejaremos posibilidad de configurar las extensiones de archivos a escanear, esto se fija en la directiva *pattern* del archivo /etc/squid*/etc/squid/SCAVR.conf.* El valor *all* fuerza a procesar todos los archivos sea cual sea la extension.

Caché de Squid limpia: No daremos la posibilidad de usar proxy Squid para descargar los archivos a la caché, escanearlos y posteriormente servirlos o no según el resultado del escaneo, así pretendemos que la caché quede limpia de virus. Esto implica que los parámetros de la sección [Proxy] del archivo de configuración del redirector no aparezcan o aparezcan comentados.

Página de alerta de virus: Fijaremos también la URL a la que redireccinar a los clientes cuando se encuentre un fichero infectado, *virusurl = http://localhost/clamAV/infovir.php.* Esto implica que debemos dejar la pagina de información en el servidor apache local y hacerla accesible a los usuarios de squid.

8.6. Descripción del Módulo Firewall (iptables + fireHOL)

8.6.1. Introducción

Un filtro de paquetes es un software que examina las cabeceras de los paquetes según van pasando, y decide que hacer con el paquete completo. Podría decidir descartarlo (DROP), como si nunca lo hubiera recibido; aceptarlo (ACCEPT), dejar que pase; u otras opciones más complejas como por ejemplo rechazarlo (REJECT).

Nosotros superpondremos una capa de abstracción a iptables, fireHOL, esto simplificará el trabajo. A lo largo de esta sección se aclarará que es fireHOL y cual es su funcionalidad.

8.6.2. Descripción de iptables y fireHOL

netfilter/iptables

Netfilter e iptables son bloques de una estructura incluida en los núcleos 2.4.x y 2.6.x de Linux. Esta estructura facilita el filtrado de paquetes, la traducción de direcciones (y puertos) de red (NA[P]T) y otras funciones de gestión de paquetes. Es fruto del rediseño y mejora de los sistemas ipchains (2.2.x) y ipfwadm (2.0.x).

Netfilter es una área de trabajo general dentro del núcleo, a la que pueden conectarse otras cosas (como el modulo de iptables). Esto significa que se requiere un núcleo más reciente que el 2.3.15 y compilar el núcleo con la opción de netfilter activada.

La herramienta iptables se comunica con el núcleo y le dice que paquetes filtrar. Iptables inserta y elimina reglas de la tabla de filtrado de paquetes del núcleo.

La configuración que tiene el cortafuegos en un momento dado está almacenada en el núcleo, esto significa que cualquier cosa que se establezca, se pierde al reiniciar el sistema. Según dice el propio *Paul "Rusty" Russell* (autor inicial), en el *Packet Filtering HOWTO*, en su lista de cosas por hacer está escribir iptables-save e iptables-restore. Mientras tanto recomienda escribir un script de inicio para configurar las reglas.

Sin embargo, nuestro propósito es crear una interfaz de configuración, y sería tedioso tener que recorrer el script de reglas de iptables. La solución que adoptaremos es superponer una capa de abstracción a iptables, esta es fireHOL. La herramienta fireHOL guarda la configuración en el archivo *firehol.conf* y la carga al arrancar el servicio, este archivo es más fácil de leer y modificar que un script.

Puede obtenerse información detallada sobre netfilter/iptables en su web oficial `http://www.netfilter.org/`.

Flujo de paquetes

El núcleo comienza con tres listas de reglas en la tabla de filtros, estas listas se llaman cadenas, son INPUT, OUTPUT y FORWARD.

Cuando un paquete alcanza un círculo en el diagrama, se examina esa cadena para decidir la suerte del paquete. Si la cadena dice que hay que descartar (DROP) el paquete, se elimina en ese mismo punto, pero si la cadena dice que hay que aceptarlo (ACCEPT), continúa su camino por el diagrama.

Una cadena es una lista de reglas. Cada regla indica «si el paquete se parece a esto, entonces esto otro es lo que hay que hacer con él». Si la regla no se ajusta al paquete, entonces se consulta la siguiente regla en la lista. Al final, si no hay más reglas por consultar, el núcleo revisa la política de la cadena para decidir qué hacer. En un sistema consciente de la seguridad, esta política suele indicar al núcleo que descarte (DROP) el paquete.

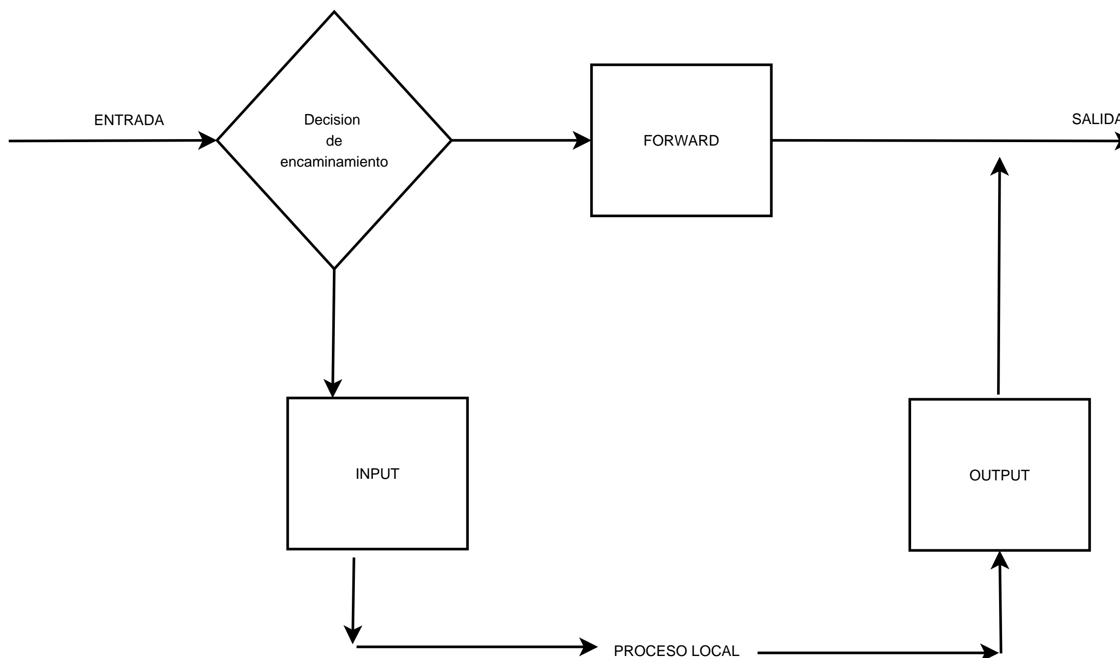

Figura 6: Iptables – Flujo de datos

Cuando llega un paquete el núcleo mira primero su destino: a esto se le llama «encaminamiento» (routing). Si está destinado a esa misma máquina, el paquete entra en el diagrama hacia la cadena INPUT. Si pasa de aquí, cualquier proceso que esté esperando por el paquete, lo recibirá. En caso contrario, si el núcleo no tiene las capacidades de reenvío activadas, o no sabe hacia dónde reenviar el paquete, se descarta. Si está activado el reenvío, y el paquete está destinado a otra interfaz de red (si tenemos otra), entonces el paquete pasa directamente a la cadena FORWARD de nuestro diagrama. Si es aceptado, entonces saldrá de la máquina. Finalmente, si un programa que se ejecuta en la máquina puede enviar paquetes de red. Estos paquetes pasan por la cadena OUTPUT de forma inmediata: si los acepta (ACCEPT), entonces el paquete continúa hacia fuera, dirigido a la interfaz a la que estuviera destinada.

En el gráfico 6 se trata de aclarar algo estas ideas.

8.6.3. fireHOL

FireHOL hace más fácil la configuración de un cortafuegos. FireHOL es un generador de reglas de iptables, pero no es solo eso, además es un lenguaje de descripción de reglas de filtrado.

Está pensado para ser más parecido al lenguaje natural y por lo tanto mucho más intuitivo. Con solo echar un vistazo al archivo de configuración de fireHOL podremos hacernos una idea de que hace la máquina en cuestión dentro de la red.

Otro de los motivos por los que fireHOL es útil para nosotros, es que el formato donde se encuentran definidas las reglas de filtrado nos permite leerlo y modificarlo con más facilidad que el script de iptables. Por lo tanto el añadir esta capa sobre iptables empeorará ligeramente el rendimiento del cortafuegos pero simplificará enormemente su gestión.

El funcionamiento de fireHOL se puede dividir en dos fases, ***procesado de configuración*** y ***activación del firewall***.

El procesado de configuración es algo lento ya que es llevado a cabo por *BASH*, no obstante esta lentitud no afecta a la seguridad del cortafuegos, ya que este sigue corriendo durante esta fase con

la configuración anterior.

Durante la fase de procesado de configuración fireHOL producirá una lista de comandos de iptables para ser ejecutados en la fase de activación, uno a uno después de haber limpiado las reglas configuradas anteriormente.

Se puede obtener más información sobre fireHOL en: `http://firehol.sourceforge.net/`. FireHOL se distribuye bajo licencia GPL, lo que permite integrarlo en nuestro proyecto sin problemas.

En el anexo F.1 de esta memoria, podemos encontrar descrito el proceso de instalación del software firehol.

También se remite al lector a dicho anexo (F.1 en la página 112) para encontrar información sobre las distintas formas de invocar a firehol. En nuestro caso los metodos de invocación que usaremos son (start, stop, restart y status). Comentamos estos a continuación y el resto en el anexo.

start Activa la configuración del firewall, que debe encontrarse en /etc/firehol/firehol.conf.

stop Para el filtrado del firewall, todo el trafico pasará sin ser comprobado.

restart es un alias de **start** para mantener la compatibilidad con /etc/init.d/iptables.

status Muestra el estado del firewall que está activo. Similar a */sbin/iptables -nxvL | less*

Lenguaje

FireHOL trata de ser un lenguaje de descripción de reglas simple e intuitivo. Antes de detallar cada uno de los comandos, acciones, parámetros y variables, veremos algunos ejemplos simples con el fin de comprobar la sencillez de fireHOL. En la tabla 3 en la página siguiente podemos verlos.

Hemos de resaltar un punto importante, las reglas dadas se ajustan solo a una dirección del tráfico, *la petición*. No se dice nada de las respuestas, fireHOL se encarga automáticamente de generar las sentencias de iptables necesarias para tratar las respuestas según la política adoptada para las peticiones.

En el anexo F.2 (página 113). En este mismo anexo también se puede encontar una descripción detallada de los comandos (router, interface, etc), y subcomandos (server, client, etc) de fireHOL, así como de las acciones (accept, reject, deny, etc), servicios y parametros opcionales de las reglas.

8.6.4. Reflexión sobre los parámetros

Nivel de log: Tras la instalación por defecto de fireHOL muchos usuarios se quejan de la aparición constante de lineas de log en la consola. A continuación tratamos de explicar a que se debe esto y como evitarlo.
FireHOL registra ("logea") el tráfico, exactamente de la misma manera que lo hace iptables. Podemos controlar la manera en que funciona el sistema de archivado de logs cambiando las variables *FIREHOL_LOG_MODE*, *FIREHOL_LOG_OPTIONS* y *FIREHOL_LOG_LEVEL*.
Para evitar que los paquetes logeados se muestren por consola debemos: configurar klogd para que considere solo el tráfico importante y configurar el nivel de log de fireHOL a un nivel no tan importante.
klogd es el kernel log daemon, un demonio del sistema que intercepta y registra los mensajes del kernel.
Para evitar que los paquetes logeados aparezcan en la consola, el nivel de klogd debe ser MENOR que el de iptables. Mostramos una tabla con estos niveles en el cuadro 4 en la página 50.

Cuadro 3: ejemplos fireHOL

Lenguaje natural	FireHOL
Mi host ofrece a Internet: - un servidor de correo - un servidor web - un servidor ftp - un servidor ssh, pero solo para el pc de mi oficina.	*office="my_office_pc.example.com"* interface eth0 lan interface ppp+internet *server smtp accept* *server http accept* *server ftp accept* *server ssh accept src $office*
Mi sistema es además una estación de trabajo y debe funcionar cualquier cliente que quiera.	office="my_office_pc.example.com" *interface eth0 lan* *interface ppp+internet* server smtp accept server http accept server ftp accept server ssh accept src $office *client all accept*
Mi LAN es segura, cualquier pc de la LAN debe poder acceder a servicios de mi host.	*office="my_office_pc.example.com"* interface eth0 lan policy accept interface ppp+internet *server smtp accept* *server http accept* *server ftp accept* *server ssh accept src $office* client all acept
Quiero que los pc's de mi LAN usen mi host como pasarela para conectarse a Internet, como clientes, a cualquier servicio que quieran.	*office="my_office_pc.example.com"* interface eth0 lan interface ppp+internet *server smtp accept* *server http accept* *server ftp accept* *server ssh accept src $office* client all acept *router lan2internet inface eth0 outface ppp+* *route all accept*
Los pc's de mi LAN tiene direcciones IP privadas por lo que tengo que hacer masquerade si quiero darles acceso a internet.	office="my_office_pc.example.com" *interface eth0 lan* *interface ppp+internet* server smtp accept server http accept server ftp accept server ssh accept src $office client all acept router lan2internet inface eth0 outface ppp+ *masquerade* route all accept

Prioridad	klogd	iptables	Descripción
0	0	emerg	El sistema está inopertativo
1	1	alert	Deben tomarse acciones de inmediato
2	2	crit	Condiciones críticas
3	3	error	Condiciones de error
4	4	warning (default)	Condiciones de aviso
5	5	notice	Condiciones normales pero significativas
6	6	info	Informativas
7	7 (por defecto)	debug	Mensajes del nivel de depuración

Cuadro 4: Prioridades de log en klogd, iptables, fireHOL

Por lo tanto para solucionar esto, incluimos la linea *FIREHOL_LOG_LEVEL="6"* en el archivo de configuración de fireHOL, /etc/firehol/firehol.conf, y bajando el nivel de klogd a 5, incluyendo la linea: *KLOGD="-c 5"*, en el archivo */etc/init.d/klogd.*

Incluir reglas de otros archivos: como ya se ha comentado, el fichero de configuración de FireHOL es un script normal de BASH. Como tal, podemos usar todas las funcionalidades de BASH en dicho archivo de configuración, esto incluye funciones, bucles, variables, I/O, etc, etc.
Así pues y haciendo uso de lo anterior, podemos definir conjuntos de reglas en otros ficheros para después incluirlas en *firehol.conf* a modo de includes. Aclaremos esto con un ejemplo, supongamos que tenemos en un archivo, firehol-squid.inc, las reglas que debemos incluir en el cortafuegos derivadas de la configuración de squid actual. Para incluir estas reglas de forma *limpia,* incluimos en el archivo en la zona adecuada del archivo de configuración firehol.conf la siguiente linea:

```
$(cat ./firehol-squid.inc)
```

Políticas por defecto: ya se ha explicado en anteriores puntos de esta memoria que la política por omisión recomendada para un cortafuegos es: bloquear todo lo que no sea explicitamente permitido. Esta será la configuración inicial de nuestro sistema. Sin embargo se dará al usuario la opción de cambiar este parámetro desde la interfaz de configuración desarrollada.

8.7. Descripción del Módulo Detector de Intrusiones IDS. (Snort)

8.7.1. Introducción

A pesar de que un enfoque clásico de la seguridad de un sistema informático siempre define como principal defensa del mismo sus controles de acceso (desde una política implantada en un cortafuegos hasta unas listas de control de acceso en un router o en el propio sistema de ficheros de una máquina), esta visión es extremadamente simplista si no tenemos en cuenta que en muchos casos esos controles no pueden protegernos ante un ataque.

Por poner un ejemplo sencillo, pensemos en un firewall donde hemos implantado una política que deje acceder al puerto 80 de nuestros servidores web desde cualquier máquina de Internet; ese cortafuegos sólo comprobará si el puerto destino de una trama es el que hemos decidido para el servicio HTTP, pero seguramente no tendrá en cuenta si ese tráfico representa o no un ataque o una violación de nuestra política de seguridad. Por ejemplo, no detendrá a un atacante que trate de acceder al archivo de contraseñas de una máquina aprovechando un *bug* del servidor web.

Llamaremos intrusión a un conjunto de acciones que intentan comprometer la integridad, confidencialidad o disponibilidad de un recurso.

Analizando esta definición, podemos darnos cuenta de que una intrusión no tiene por qué consistir en un acceso no autorizado a una máquina, también puede ser una negación de servicio.

A los sistemas utilizados para detectar las intrusiones o los intentos de intrusión se les denomina sistemas de detección de intrusiones (*Intrusion Detection Systems*, IDS) o, más habitualmente, sistemas de detección de intrusos. Cualquier mecanismo de seguridad con este propósito puede ser considerado un IDS, pero generalmente sólo se aplica esta denominación a los sistemas automáticos (software o hardware).

8.7.2. Descripción de Snort

Snort es un sistema de detección de intrusos distribuido con licencia GPL, capaz de analizar tráfico en tiempo real y registrar (loguear) paquetes en redes IP. Puede realizar análisis de protocolos y contenidos, y puede detectar una gran variedad de ataques y pruebas tales como *buffer overflows*, escaneos de puertos, ataques CGI, pruebas SMB y otros muchos.

Snort usa un lenguaje flexible de definición de reglas para describir que tráfico debe ser recogido y cual no, además de una mecánica de detección modular basada en *plug-ins*. También ofrece la posibilidad de generar alertas en tiempo real, incorporando mecanismos de alerta variados: syslog, un fichero especifico, socket UNIX o WinPopups para clientes Windows que usen clientes SAMBA.

Snort puede ser configurado en 3 modos de funcionamiento básicos:

- **Sniffer:** simplemente lee todos los paquetes que atraviesan la red y los muestra por pantalla en la consola.
- **Packet logger:** guarda los paquetes capturados en disco en lugar de mostrarlos por pantalla.
- **Sistema completo de detección de intrusos de red:** es la más compleja y configurable de los 3 modos de funcionamiento. Analiza el tráfico capturado y comprueba si se ajusta a las reglas configuradas por el usuario, realizando determinadas acciones en cada caso.

Para los objetivos de nuestro proyecto, nos interesa que Snort funcione en el tercero de los modos, NIDS (*Network Intrusion Detection System*), de manera que no se guarde en disco cada paquete de la red.

Snort acepta un gran número de parámetro por la linea de comandos, permitiendo ajustar su uso completamente a nuestras necesidades.

De forma orientativa, en el anexo G.1 en la página 117 mostramos una relación de opciones que puede recibir Snort.

Nosotros arrancamos Snort como demonio, *"/etc/init.d/snort start"*. Veamos el proceso que corre en nuestro sistema con una configuración dada a modo de ejemplo:

```
/usr/sbin/snort -m 027 -D -l /var/log/snort -d -u snort -g snort -c /etc/snort/snort.conf -S HOME_NET=[any]
-i eth0
```

Para que corra como demonio incluimos la opción ***-D***; el directorio de logs estará en su ruta por defecto (***-l /var/log/snort***); Snort correrá como usuario y grupo *snort,* (***-u snort -g snort***); recogerá la configuración del fichero de configuración por defecto de Snort, ***(-c /etc/snort/snort.conf)***; Y escuchará en el interfaz eth0 ***(-i eth0)***.

A continuación resumiremos el funcionamiento interno de Snort: Snort adquiere los paquetes en crudo directamente de la interfaz de red. La adquisición de paquetes es llevada a cabo por libpcap, que es una biblioteca externa a Snort. ***Libpcap*** es portable a casi todas las plataformas actuales, haciendo a Snort realmente una aplicación independiente de la plataforma.

El ***preprocesador*** es el primer componente interno de Snort que un paquete capturado se encuentra, su propósito es "destripar" las cabeceras de los paquetes. Funciona descifrando la pila de protocolo TCP/IP y colocando los paquetes en una estructura de datos. Después los paquetes se encaminan a los preprocesadores.

Los preprocesadores de Snort realizan dos tareas fundamentales: manipulan los paquetes para que el motor de detección pueda analizarlos correctamente y analiza el tráfico en busca de elementos sospechosos que no se pueden descubrir simplemente inspeccionando los patrones.

Snort cuenta con un gran número de preprocesadores, la mayoría de los cuales se han agregado para combatir nuevos métodos de evasión de los IDS. Todo, desde shellcode polimórfico hasta paquetes fragmentados, puede ser detectado por los preprocesadores de Snort. Después de que el tráfico atraviese los preprocesadores se envía al motor de detección.

El ***motor de detección*** es el responsable de la detección de patrones peligrosos (signature detection). Las reglas de detección de Snort se cargan en el motor de detección y se clasifican en una estructura de datos tipo árbol. Esta estructura en forma de árbol está implementada para ser lo más eficiente posible, de manera que se minimicen el número de pruebas a realizar para descubrir actividades maliciosas. Después de detectar estas actividades maliciosas, Snort escribe los datos de intrusiones en una amplia variedad de salidas.

Los ***plugins de la salida*** son los medios que Snort tiene para mostrar los datos del motor de detección. Snort se puede configurar con múltiples plugins de salida para facilitar la gestión los datos de intrusión. Los plugins de salida permiten mostrar los resultados según un formato sencillo delimitado por comas, hasta la inclusión de los datos en una base de datos.

En la figura 7, podemos ver gráficamente este proceso. Para obtener más información acerca Snort podemos visitar: `http://www.snort.org`.

La versión 3.1 de Debian GNU/Linux (Sarge) cuenta con paquetes precompilados de Snort en sus repositorios. Por tanto su instlación es sencilla. En el anexo G.2 de esta memoria, podemos ver en detalle el proceso de instalación y configuración del IDS Snort.

Vemos que se instalará también un paquete de reglas desarrollado por la comunidad Snort totalmente funcionales: *snort-rules-default.* Estas reglas pueden servir de base para desarrollar reglas propias más complejas o ajustarlas a las necesidades de cada escenario.

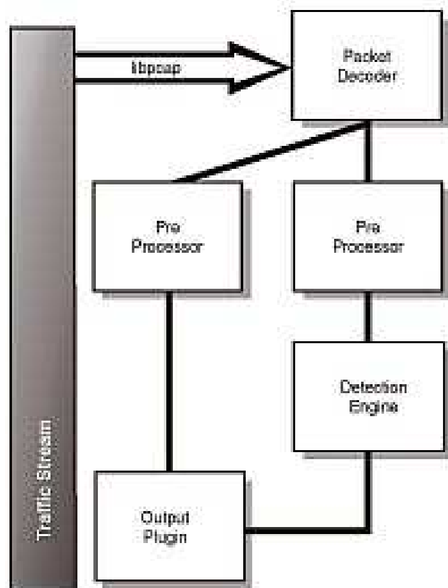

Figura 7: Componentes de Snort

Snort tiene diversos archivos de configuración en el directorio */etc/snort/,* así mismo el fichero de alertas "alert" y otros ficheros generados como resultado por Snort pueden encontrarse en */var/log/snort/.*

Entre los archivos de configuración destacaremos ***snort.conf***, donde está la configuración básica de Snort. El archivo de configuración de Snort, */etc/snort/snort.conf* tiene cuatro partes bien diferenciadas:

- Variables de red.
- Configuración de preprocesadores.
- Configuración de plugins de salida.
- Personalización de reglas del motor de detección.

Comentamos más en profundidad su contenido en el anexo G.3 de esta memoria de proyecto.

snort.common.parameters y ***snort.debian.conf***, que contienen las opciones comunes y específicas de Debian según las cuales se ejecuta el demonio de Snort.

Otros archivos son ***classification.config*** en el que se asigna prioridad a cada regla; ***threshold.conf***, que se configura para reducir el número de alertas de reglas demasiado ruidosas, se define el número de veces que una alerta es logeada en un intervalo de tiempo; y ***reference.config*** donde se definen las URL's a las que se hace referencia en las reglas.

Por último en el directorio ***/etc/snort/rules/***, encontraremos las reglas "snort-rules", que fijarán el funcionamiento de snort. Si hemos instalado el paquete ***snort-rules-default***, en este directorio encontraremos un conjunto de ficheros que contienen estas reglas. Podemos encontrar la lista de estos ficheros en el anexo G.2 en la página 118.

Como ejemplo copiamos aquí un fragmento del contenido de uno de estos archivos, chat.rules:

```
# (C) Copyright 2001-2004, Martin Roesch, Brian Caswell, et al.
# All rights reserved.
# $Id: chat.rules,v 1.22.2.2 2004/08/10 13:52:05 bmc Exp $
#————-
# CHAT RULES
#————-
# These signatures look for people using various types of chat programs (for
# example: AIM, ICQ, and IRC) which may be against corporate policy
alert tcp $HOME_NET any ->$EXTERNAL_NET any (msg:"CHAT ICQ access"; flow:to_server,established; content:"User-Agent|3A|ICQ"; classtype:policy-violation; sid:541; rev:9;)
alert tcp $EXTERNAL_NET 80 ->$HOME_NET any (msg:"CHAT ICQ forced user addition"; flow:established,to_client; content:"Content-Type|3A| application/x-icq"; nocase; content:"[ICQ User]"; reference:bugtraq,3226; reference:cve,2001-1305; classtype:policy-violation; sid:1832; rev:7;)
alert tcp $HOME_NET any <>$EXTERNAL_NET 1863 (msg:"CHAT MSN message"; flow:established; content:"MSG "; depth:4; content:"Content-Type|3A|"; nocase; content:"text/plain"; distance:1; classtype:policy-violation; sid:540; rev:11;)
alert tcp $HOME_NET any <>$EXTERNAL_NET 1863 (msg:"CHAT MSN file transfer request"; flow:established; content:"MSG "; depth:4; content:"Content-Type|3A|"; distance:0; nocase; content:"text/x-msmsgsinvite"; distance:0; nocase; content:"Application-Name|3A|"; content:"File Transfer"; distance:0; nocase; classtype:policy-violation; sid:1986; rev:4;)
alert tcp $HOME_NET any <>$EXTERNAL_NET 1863 (msg:"CHAT MSN file transfer accept"; flow:established; content:"MSG "; depth:4; content:"Content-Type|3A|"; nocase; content:"text/x-msmsgsinvite"; distance:0; content:"Invitation-Command|3A|"; content:"ACCEPT"; distance:1; classtype:policy-violation; sid:1988; rev:3;)
```

Como vemos el formato de definición de reglas, a pesar de ser altamente flexible y potente, no es muy amigable a primera vista.

Cada regla de Snort está dividida en dos secciones lógicas: la cabecera de la regla y las opciones de la regla.

La cabecera contiene la acción, el protocolo, las direcciones IP origen y destino, la máscara de red y los puertos origen y destino.

Las sección de opciones contiene, mensajes de alerta e información de la parte del paquete que hay que inspeccionar para decidir si la acción de la regla debe realizarse.

8.7.3. Reflexión sobre los parámetros y clasificación de estos

Gestión de reglas: En nuestro caso, la interfaz de configuración no tratará en profundidad la configuración de este módulo. Fundamentalmente se gestionarán los ficheros de reglas, pudiendo importarse y editarse. Por lo tanto tan solo la última parte del archivo de configuración *snort.conf*, nos será de interés. Las directivas *include,* que toman como parámetros la ruta a los archivos de reglas, son los que deberemos procesar para obtener la ruta a los archivos que contienen las reglas.
Los archivos que contengan las reglas, deben tener la extensión *.rules*, y estarán en el directorio */etc/snort/rules,* por lo tanto debemos darle el valor anterior a la variable *$RULE_PATH.*

Variables de red: dejaremos la mayoría de estas directivas fijas a su valor por defecto, excepto alguna como *HOME_NET* a la que podemos asignar el valor de la ip nuestra red interna: *var HOME_NET redinterna.*

Plug in de salida: nos inclinamos a usar el plugin de salida *fast_alert*, esto implica incluir la linea siguiente en *snort.conf:*

```
output alert_fast: alert.fast
```

Representación de estadísticas: es interesante disponer de alguna aplicación que genere estadísticas a partir de los datos generados por Snort, estos datos quedan en el directorio */var/log/snort/*. Existen múltiples herramientas que pueden hacer esto, podemos encontrar algunas en el siguiente enlace: `http://www.snort.org/dl/contrib/data_analysis/` .
La mayoría de estas herramientas requieren una base de datos para trabajar, desde el principio hemos tratado de evitar por todos los medios la integración de una base de datos en nuestro proyecto, para evitar cargar un servidor de seguridad cuya función en la red es crítica. Por lo tanto no es el momento de decidirnos a incluirla, más aun si tenemos en cuenta que la representación de datos estadísticos es una funcionalidad secundaria dentro del proyecto.
A partir de la reflexión anterior nos decidimos por la aplicación ***SnortALog***, script en pearl distribuido con licencia GPL, capaz de generar informes en html, PDF y formato texto, con gráficos GIF, PNG o JPG, y capaz de trabajar a partir de distintos plugins de salida de Snort, entre ellos el que nos interesa alert_fast.

8.8. Diseño de interfaces

En este apartado de la memoria del proyecto se profundiza en el diseño de las interfaces de la aplicación. Esta fase de diseño cobra gran importancia ya que las decisiones tomadas en ellas afectarán al futuro desarrollo de nuestra aplicacion.

Convenciones para la descripción del diseño gráfico de las interfaces:

Se ha intentado que el diseño gráfico empleado aquí sea lo más "genérico" posible. Se ha prescindido de todo tipo de *abusos* a nivel gráfico, manteniendo una *austeridad* más que evidente en este apartado. La razón es que se intenta mostrar de una forma lo más clara posible la *estructura* de las interfaces, que es lo realmente importante es esta fase del diseño.

Si bien la presentación final de las interfaces, tal y como las verá el usuario, no tienen por qué ser exactamente igual a las mostradas aquí, su "estructura" sí coincidirá. Es decir, los elementos que se muestran, el tipo de los formularios, etc. coincidirá a nivel estructural y de distribución espacial en las interfaces de usuario; su presentación (gráficos, colores, tipos de letra, tamaño del texto, forma de los botones, etc.) podrá variar, dependiendo de la *plantilla* que finalmente se utilice. Esta es la ventaja de separar la presentación del control (se puede variar libremente uno, sin realizar modificaciones en el otro).

Con esto dicho, consideraremos que el usuario interactúa con el sistema gráfico de configuración, a través de una serie de *formularios*. Estos, entendidos de la misma manera que los *formularios Web*, ya que en la implementación se utilizarán estos últimos para plasmar en el sistema real el diseño aquí presentado. Por tanto su organización, estructura, tipos de elementos, etc. serán los mismos que se pueden encontrar en los formularios Web descritos en la especificación *XHTML 1.0*.

En estos formularios, el usuario introducirá valores de configuración mediante campos editables disponibles al efecto. El usuario se podrá desplazar de un formulario a otro mediante un sistema de menús. Los formularios estarán organizados en estos menús en grupos, uno correspondiente a cada módulo. De esta forma, queda una visión de conjunto (relaciona unos formularios con otros) para el propósito que se persigue (la configuración de un módulo de forma aislada[2]).

Existirán dos tipos de formularios:

- Formularios *"principales"*: son formularios normales, independientes de los demás.
- Formularios *"auxiliares"*: son formularios que son lanzados por los *"principales"*, para alcanzar un nivel de detalle superior o realizar una acción de configuración extendida con respecto a la que se puede realizar en su formulario *"principal"* asociado.

Diferenciaremos los formularios en el diseño de la siguiente manera: los formularios "principales" tendrán una anchura completa de página, y una barra de título de color oscuro (azul); los formularios "auxiliares" tendrán una anchura menor, una barra de título de color claro (amarillo), y en su título mostrarán, a modo de *prefijo*, el nombre del formulario "principal" al que pertenecen.

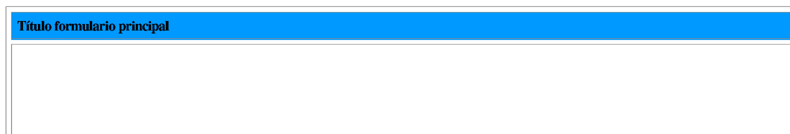

Figura 8: Formulario "principal"

[2] En realidad, esto es una ilusión, sencillamente para "comodidad" del usuario (que acota el alcance de sus acciones), pero realmente en la práctica ningún módulo se configura enteramente independiente de los demás, debido al mecanismo de comunicación entre módulos mencionado.

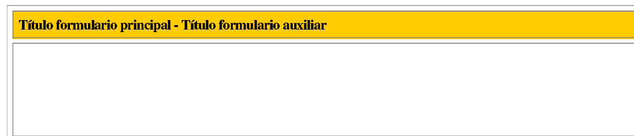

Figura 9: Formulario "auxiliar"

Por tanto, tenemos que un *'formulario'* es un bloque o recuadro, con un título. Dentro de dicho bloque, además, se encontrarán los controles (campos editables, etc.) con los que el usuario podrá interactuar para comunicarse con el sistema.

Debajo de todos los formularios (si bien en algún momento no aparecerán, aunque asumiremos que están ahí), aparecerán en la parte inferior dos botones, con las etiquetas *"Guardar configuración"* y *"Restablecer configuración"*. Se encuentran fuera del recuadro ya que la acción que realizan engloba al conjunto de los controles y datos que aglutina el formulario. El significado de dichos botones es el siguiente:

- **Guardar configuración:** Almacena en el sistema las opciones de configuración que el usuario ha introducido por medio del formulario.
- **Restablecer configuración:** *"Recarga"* el formulario, restaurando los valores de los campos al que hay almacenado en el sistema (desechando por tanto las modificaciones que el usuario ha realizado sobre los mismos, y no ha guardado).

En realidad, estos botones podrán aparecer bajo un formulario (organizado por 'solapas'), o bajo el conjunto de todos ellos, dependiendo de cómo se realice el diseño final de las plantillas de los formularios Web. En todo caso, la acción representada por estos botones afectará *siempre* al conjunto de todos ellos. Por simplicidad, asumiremos que cada formulario se presenta de manera independiente (de forma que en el diseño veremos sólo un formulario sobre los dos botones mencionados).

Como ya hemos dicho, los controles y elementos que aparecen en el interior de los formularios serán los habituales en un formulario Web, de los posibles según la especificación *XHTML 1.0* (así tenemos: botones, campos de texto, casillas de verificación —*checkbox*–, botones radiales –*radio button*–, etc). En en diseño, también se verán en múltiples ocasiones, repartidas por el formulario, ciertas cadenas de texto entre llaves '{}' que conforman *etiquetas* (que serán: texto descriptivo de los controles, formatos de valores introducibles, comentarios, información de estado, o cualquier valor que será variable y por tanto se *renderizará* de forma dinámica en el momento de presentar el formulario). Por tanto, en los formularios tendremos etiquetas que se presentarán de dos formas, según sean:

- **Etiquetas estáticas:** (color negro, texto de formato normal) Se presentarán siempre tal cual se muestran aquí, su contenido no cambia.
- ***{Etiquetas dinámicas}:*** (color azul, texto en *itálica*, encerrado entre llaves) Su contenido es variable, y podrá venir dado por diferentes causas u orígenes de datos (por ejemplo, será un valor de configuración obtenido del sistema, una secuencia de valores, la respuesta a la ejecución de un comando, etc.)

El formato y/o valores posibles de dichas etiquetas dinámicas vendrán descritas en una *"leyenda"* anexada al formulario (que sólo veremos en el diseño, es decir, el formulario una vez se muestre al usuario sustituirá estas etiquetas dinámicas por el contenido que les corresponda en ese momento dado). En dicha leyenda, se *asignarán* valores posibles a las etiquetas, ya sean conjuntos de cadenas estáticas (podrá aparecer una u otra), una especificación de formato (valor numérico o alfanumérico), combinaciones de estos, etc... Por ejemplo:

{Estado} = { "Activo" , "Parado" }
{Nombre} = { AAAAAAAAAAAA... }
{Dirección IP} = { NNN.NNN.NNN.NNN *(formato de dirección IP)* }
{Regla Ponderada} = { AAAA...=NNNN... }

Con esta leyenda se indica que: *{Estado}* tendrá dos valores posibles: las cadenas "Activo" o "Parado"; *{Nombre}* tendrá un valor alfanumérico de longitud variable; *{Dirección IP}* podrá tener únicamente un formato de números de 3 cifras (máximo) separados por puntos (es decir, el formato de una dirección IP); *{Regla Ponderada}* especifica un formato de asignación del tipo *nombre=valor*, etc...

Con el fin de no dificultar la lectura de esta memoria con detalles de diseño de los formularios, específicos de cada módulo, estos detalles se presentan en el anexo H en la página 121 de esta memoria.

En este anexo se expone para cada módulo lo siguiente:

Explicación detallada de los formularios. Se explican uno a uno los elementos de los formularios de los módulos. Sus posibles valores y su función. También se puede encontrar en este anexo, imágenes de todos y cada uno de los formularios incluidos en la aplicación.

Reflexión acerca de los formularios de configuración de cada módulo. Se justifica la inclusión de cada uno de los elementos del formulario.

Relación con los parámetros de configuración del software. Se presentan tablas en las que se relacionan los elementos de cada formulario con los parámetros de configuración de cada software (configurado por nuestra aplicación).
Al final de esta sección, presentamos a modo de ejemplo una de estas tablas. En este caso se trata de los elementos de configuración básica el módulo Squid. En la primera columna se muestran los campos del formulario, en la segunda los parámetros, del archivo squid.conf, relacionados con estos campos y por último una tercera columna con los comentarios necesarios.

Importación/Exportación de parámetros. Se indican los parámetros *importables* (parámetros que puede necesitar un módulo de otros módulos para ajustar su funcionamiento, o para hacerlo compatible con la funcionalidad común que se persigue) o *exportables* (parámetros que ofrecerá un módulo a otros módulos que necesiten comunicarse con él, o simplemente para evitar al usuario la introducción de un mismo dato dos veces, en distintos módulos).
A continuación vemos un ejemplo de estas tablas. En este caso se trata de los parámetros importables y exportables del módulo filtro de contenidos, DansGuardian.

Ejemplo tabla reflexión de parámetros de módulo

Campo	**Parámetros**	**Comentarios**
Puerto de escucha	`http_port`	Sólo se establecerá el puerto, no la dirección IP.
Modo Transparente	`httpd_accel_host` `httpd_accel_port` `httpd_accel_with_proxy` `httpd_accel_uses_host_header`	Se introducirán en estos parámetros los puertos de navegación especificados en *Control de acceso – Puertos permitidos*. Será necesario comunicar al Módulo Cortafuegos también este dato (ver la reflexión anteriormente comentada).

Ejemplo de tabla de importación de parámetros

Módulo	**Parámetro**	**Descripción**
Proxy (Squid)	`puerto_escucha`	Se empleará este valor del Proxy Squid para que, en caso de que el usuario seleccione '*Enlazado con el Proxy – Proxy Squid local*' se sepa automáticamente a qué puerto conectarse (para el parámetro `proxyport`).
Proxy (Squid)	`activado`	Si el Proxy Squid local está desactivado, en caso de que el usuario seleccione '*Enlazado con el Proxy – Proxy Squid local*' no se configurará valor alguno en el parámetro `proxyport`.

Ejemplo de tabla de exportación de parámetros:

Parámetro	**Descripción**
`activado`	Se establecerá a `1` en el caso de que el módulo se haya configurado para activarse en el arranque (formulario **"Control del módulo"**), o al valor `0` en el caso contrario.
`puerto_escucha`	Puerto en el que `dansguardian` recibirá las peticiones HTTP. Será el valor introducido en el campo ***Puerto de escucha*** del formulario **"Configuración básica"**.
`proxy_externo`	Tendrá el valor `1` en el caso de que se haya configurado un Proxy externo al que conectarse (***Enlazado con el Proxy – Especificar otro*** del formulario **"Configuración básica"**), ó el valor `0` en caso contrario.
`direccion_proxy_externo`	Dirección IP (o nombre de host) del proxy externo configurado por el usuario. Se tomará del campo ***Enlazado con el Proxy – Dirección IP*** del formulario **"Configuración básica"**.
`puerto_proxy_externo`	Puerto configurado para el proxy externo (se toma del campo ***Enlazado con el Proxy – Puerto*** del formulario **"Configuración básica"**).

9. Implementación

9.1. Introducción

Veremos qué toma de decisiones se ha efectuado a la hora de realizar la Implementación del sistema, mostrando el Modelo de Clases del Sistema, las librerías estándares que se han utilizado, así como las decisiones más importantes a nivel arquitectónico que describen cómo se han implementado ciertas cuestiones a nivel general (y en algunos casos en concreto dignos de mención).

9.2. Modelo de Clases del sistema

Para entender mejor el sistema, primero mostraremos un modelo que lo represente desde un punto de vista no detallado. Este modelo se realizó a partir de los requisitos del sistema (ver sección 7.1 en la página 27), ya que estos, y en especial el **RNF4**, dan ya de por sí cierta información acerca de cómo se deberá construir el sistema.

Para representar el sistema se han identificado los elementos de los que constaría y, modelizándolos mediante Clases, se ha llegado al siguiente Diagrama de Clases del Sistema, que muestra las diferentes clases identificadas, y las relaciones entre ellas.

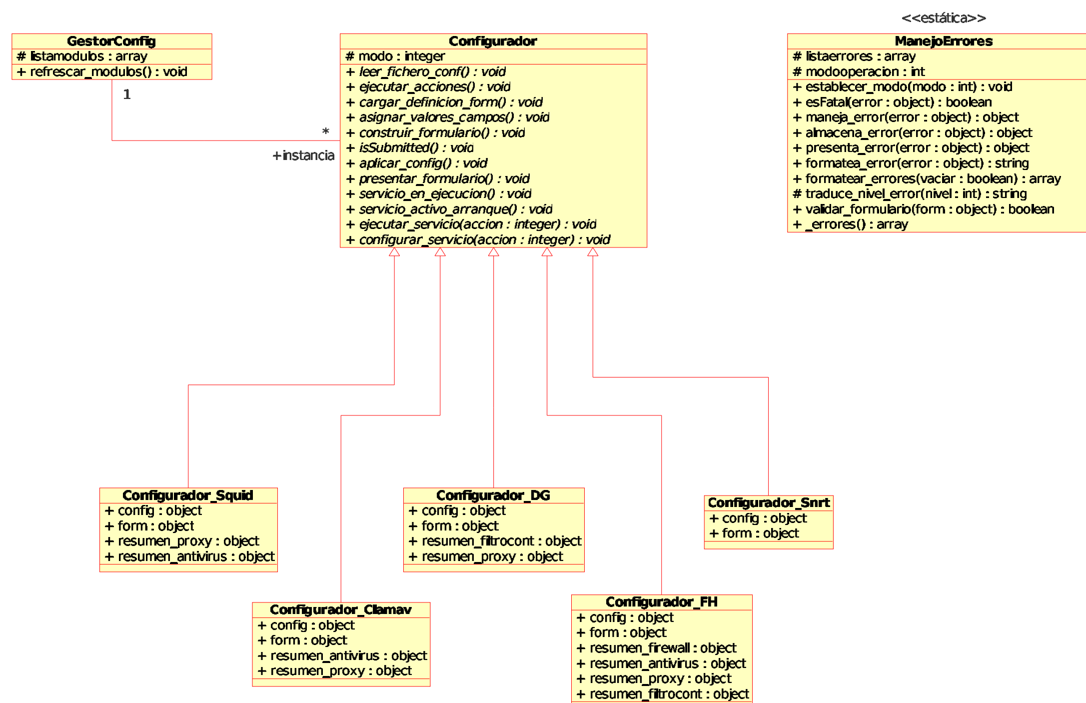

Figura 10: Modelo de Clases del Sistema

Descripción

Como se puede ver en este diagrama, se tiene una clase principal "**Configurador**", de la cual heredan una serie de clases hijas. Estas clases hijas tienen como misión concretizar la implementación

para cada caso en concreto. Además, se muestran otras dos clases que se han implementado en el sistema, y que son "GestorConfig" y "ManejoErrores". Sobre estas dos clases se hablará en más detalle posteriormente.

La clase "Configurador" es una abstracción del proceso de diálogo entre el Usuario y el Módulo que desea configurar (si bien, también podría entenderse como el módulo en sí mismo).

Las restantes clases hijas, que heredan de "Configurador", tienen como misión implementar el funcionamiento de la clase para cada módulo en concreto, esto es: **Configurador_Squid** para el módulo del *Proxy Squid*, **Configurador_Clamav** para el módulo del *Antivirus ClamAV*, etc... Esto permite, por ejemplo, en el futuro sustituir cualquiera de estas clases hijas por una implementación específica para otro componente de Software diferente (por ejemplo, otro antivirus que no sea el *ClamAV*), sin tener que modificar ni la implementación de los demás módulos, ni la interfaz que se emplea para dialogar con el módulo.

Justificación

Por los requisitos del sistema, como se ha visto anteriormente, se hace necesario que los diferentes módulos se comuniquen entre sí para proporcionarse mutuamente información de configuración. Esto implica que, una vez que se realicen cambios en la configuración de un módulo, los restantes deban leer estos cambios para "refrescar" su configuración. Esto se implementa mediante el paso de mensajes: el módulo con el que dialoga el Usuario, y que realiza cambios en su configuración, comunica a los demás módulos que deben refrescar su configuración (para introducir cambios en los diferentes ficheros de configuración que se ven afectados).

Esta es la razón de ser del atributo "modo", de la clase padre "Configurador": cuando un módulo necesite refrescar su configuración, se instanciará en *"modo Refrescar"* ("`FW_MODO_REFRESCAR`"), en caso contrario, en *"modo Normal"* ("`FW_MODO_NORMAL`"). Esto indicará al objeto internamente en qué *"modo de funcionamiento"* se encuentra, lo cual será de especial importancia en cuanto al procesamiento de la interfaz de usuario, como veremos posteriormente. Ello es debido a que un Módulo que está en *"modo Refrescar"*, no debe procesar información alguna referente a la interfaz de usuario (ya que en ese momento el usuario está dialogando con otro módulo). Este mecanismo evitará errores, y el desarrollador sabrá a "qué atenerse" en cada situación.

Descripción de atributos y métodos

Como se puede ver en el diagrama, las clases hijas tienen en común que poseen dos atributos importantes: "config" y "form". El significado de los mismos es el siguiente:

- `config` es la representación en memoria de la estructura y contenido del fichero de configuración.
- `form` es la representación de la estructura y contenido de los campos visibles en la interfaz de usuario.

Realmente se tratan de objetos –clases–, que más tarde se verá cómo se han implementado (concretamente, utilizando bibliotecas disponibles al público, en el contexto del *Open Source*). La idea principal que los justifica es que un módulo tiene como función la transformación de los parámetros de los ficheros de configuración, en campos visibles en el formulario y configurables por el usuario, así como el proceso inverso. Por tanto debe almacenar internamente tanto el conjunto de parámetros de configuración tal y como se encuentran en el correspondiente fichero o ficheros del disco, como el conjunto de campos que contiene la interfaz de usuario (con sus tipos, restricciones, formato de visualización, etc.). De ahí la existencia de estos dos atributos.

La instanciación de cada uno de estos objetos `config` y `form`, con la estructura de datos concreta que necesite cada módulo, será responsabilidad de dicha subclase.

En cuanto a los métodos, podemos agruparlos en:

a) Métodos de configuración

- leer_fichero_conf() – lee el fichero (o ficheros) de configuración correspondientes al módulo, valida su sintaxis y semántica, e introduce su contenido en el atributo *'config'* del objeto.
- ejecutar_acciones() – ejecuta las acciones inmediatas que precise el módulo durante su configuración (como arrancar o parar servicios) o las que el usuario solicite a través de la interfaz (acciones como: vaciar la caché, actualizar el antivirus, etc...)
- cargar_definicion_form() – carga la definición del formulario (campos, tipos, apariencia) y genera la correspondiente estructura en memoria.
- asignar_valores_campos() – rellena esta estructura en memoria con los valores leídos del fichero de configuración, realizando las transformaciones que sean necesarias.
- construir_formulario() – crea el formulario gráfico que el usuario verá en pantalla, conteniendo toda la información anteriormente recopilada.
- aplicar_config() – una vez que el usuario ha introducido los valores deseados, y envía estos al sistema, se capturan los mismos, se validan y se transforman para introducirlos de nuevo en los ficheros de configuración que sea necesario.
- presentar_formulario() – se presenta finalmente el formulario al usuario, con los posibles mensajes de error fruto de las validaciones anteriores anexos al mismo.

Como se puede ver, estos módulos se podrían ejecutar de una forma prácticamente secuencial en el flujo normal de ejecución de un módulo. Esto se ha diseñado así expresamente, con el objetivo de hacer el código más sencillo y entendible, a la vez que fácilmente modificable y ampliable. Teniendo esto en cuenta, la ejecución de dichos métodos siguen el flujo 'normal' de:

Fichero de configuración → Validar y transformar → Cargar formulario → Rellenar formulario → Presentar formulario.

Pero también es válido para este otro flujo 'inverso':

Datos introducidos por el usuario → Validar y transformar → Cargar formulario → Rellenar formulario → Escribir configuración → Presentar resultados.

Como se puede ver, lo que varían son las fases iniciales y finales, siendo gran parte del resto del proceso común a ambos flujos de datos. El segundo flujo tiene en común con el primero que para poder escribir los datos modificados, es necesario leerlos antes. Dicho de otro modo, el usuario introduce modificaciones en los datos que se leen de la configuración. Por tanto el flujo lectura-proceso-escritura se deberá seguir realizando en cada caso de forma casi idéntica, ya sea sólo para mostrar datos, como para realizar modificaciones en los mismos (también es común en ambos flujos la presentación del formulario al final). Este procesamiento 'global' es lo que se implementa en los métodos enumerados anteriormente:

`leer_fichero_conf()` → `ejecutar_acciones()` → `cargar_definicion_form()` → `asignar_valores_campos()` → `construir_formulario()` → `aplicar_config()` → `presentar_formulario()`

b) Métodos de control de servicios

- servicio_en_ejecucion() – consulta si un servicio del sistema se encuentra actualmente en ejecución.
- servicio_activo_arranque() – consulta si un servicio del sistema está configurado para arrancarse en el inicio.
- ejecutar_servicio(accion) – realiza las acciones inmediatas necesarias para poner en marcha, o detener la ejecución, de un servicio del sistema.
- configurar_servicio(accion) – realiza las acciones diferidas necesarias para configurar, en el arranque del sistema, la ejecución (o no) de un servicio.

En cuanto a estos métodos, como es lógico, en cada módulo se implementarán para controlar el servicio del sistema que realiza la ejecución del componente software íntimamente relacionado con el Módulo. P.ej. en el caso del módulo del Proxy Squid, estos métodos controlarán la ejecución del servicio `squid` en el sistema.

9.3. Consideraciones sobre la arquitectura del sistema

Hasta el momento hemos ideado la estructura básica de la aplicación, pero no hemos hablado aún de cuestiones accesorias pero decisivas para realizar su implementación. Veremos más en profundidad estas cuestiones:

9.3.1. Librerías estándares utilizadas

Considerando la funcionalidad requerida, para simplificar la codificación se han elegido una serie de librerías de clases estándares existentes, extraídas de repositorios de código reutilizable publicados en Internet, puestos a disposición de la comunidad *Open-Source* por grupos de desarrollo voluntarios y sin ánimo de lucro (como suele ser común en los proyectos GPL). Éstas son:

- **PEAR::Config:** facilita el procesamiento de ficheros de configuración de texto en formatos conocidos y estándares (tales como XML, INI, o el estándar genérico "tipo Apache").
- **patForms:** librería de generación de formularios Web, y de ayuda al procesamiento de los datos introducidos a través de los mismos.
- **patTemplates:** librería para el manejo de plantillas Web, fácilmente acoplable con patForms.
- **patError:** gestión de errores en la aplicación, utilizada de forma exhaustiva por patForms.
- **patSessions:** gestión de sesiones en la aplicación, fácilmente acoplable con patForms.

Estas librerías (desarrolladas en el lenguaje de *scripts* PHP) facilitan enormemente la tarea de, por un lado leer la información de configuración a partir de los ficheros de texto de los programas antes mencionados (en este caso mediante `PEAR::Config`), y por otro generar el formulario Web (con `patForms`, y las librerías auxiliares) que presentará una representación gráfica del contenido de dichos ficheros ante el usuario, ofreciéndole la posibilidad de modificar su contenido.

En el sentido inverso de la comunicación, el usuario puede introducir los valores que desee y enviarlos de vuelta a la aplicación –a la capa de lógica de negocio–, pudiendo ser leídos con facilidad (de nuevo con la ayuda de `patForms`) y transformarlos para introducirlos posteriormente en los ficheros de configuración necesarios, con la ayuda de `PEAR::Config`.

9.3.2. Definición de formularios en XML

Como se comentó en el Análisis de Requisitos (ver 7.1), y en concreto en el requisito no funcional **RNF5**, se ha decidido emplear preferentemente el lenguaje XML para describir las estructuras de datos referentes a los interfaces de usuario (formularios de configuración).

En un compromiso entre mantener este principio y la flexibilidad que proporciona, así como no aumentar demasiado la carga del sistema en cuanto al procesamiento de los formularios y la transformación de los mismos se refiere, sin menospreciar la conveniencia de poder personalizar la apariencia de la aplicación de una forma simple, de ha optado por un esquema mixto que utiliza tanto XML como HTML para la descripción de los formularios (pantallas) de configuración del sistema, estando repartida la responsabilidad de cada lenguaje como a continuación se indica:

- HTML para las plantillas (es decir, la *apariencia*)
- XML para los elementos estructurados (independientes de la apariencia)

Es decir, que por un lado se emplearán estas *plantillas HTML* (esto es, código HTML en el que aparecen ciertas "marcas" o *etiquetas* donde se insertará el contenido dinámico posteriormente) que representan enteramente la apariencia de los formularios (y al tener una estructura interna muy similar a la que tendrán finalmente –una vez renderizados– resultan fácilmente personalizables); y por otro lado ficheros XML donde se describen los campos que aparecerán en los formularios y los metadatos asociados a ellos (es decir, datos que describen estos campos, como son por ejemplo: el tipo, longitud, nombre, descripción, atributos...)

Figura 11: Definición de los formularios mediante HTML y XML

Veremos qué posibilidades tendremos a la hora de diseñar la implementación de estos elementos.

a) Plantillas HTML

La idea es que el código HTML sea fácilmente procesable por patForms (el núcleo de procesamiento de formularios de la aplicación) de forma que éste pueda incluir los elementos dinámicos que necesite.

Tras estudiar las posibilidades existentes, decidimos el uso de **patTemplates**, que entre otras cosas es capaz de realizar sustitución de etiquetas dentro del código HTML, así como la repetición de secciones de código una o múltiples veces (lo cual es útil cuando se tiene un nº indeterminado de controles).

Su total integración con **patForms** facilita en gran medida la implementación, ya que actúa como una especie de filtro de salida o *renderizador* a la hora de generar la salida HTML, sustituyendo automáticamente las etiquetas que aparecen en la plantilla en el momento de generar el formulario.

b) Definiciones XML

Se utilizará **patForms_Definition**, una clase auxiliar de patForms que realiza, con ayuda de **PEAR::XML_Serializer**, una *serialización* y *des-serialización* de la estructura en memoria de la definición de los formularios. Esta estructura serializada está descrita en XML y por tanto nos servirá como base para cargar una definición inicial de los mismos a partir de un fichero.

Se puede ver con más detalle notas sobre la implementación concreta en la sección 9.4.3 en la página 69.

9.3.3. Gestión de errores

En una aplicación PHP sencilla, la gestión de errores suele realizarse de una manera muy simple. Cuando se produce un error, se para la ejecución del script y se presenta éste al usuario. Por ejemplo:

```
die("Se ha producido un error");
```

Esta sentencia termina la ejecución del script y muestra en la salida (página Web) el mensaje que se le pasa como argumento. En este caso presentaría en la página Web resultante la cadena de texto "Se ha producido un error". Normalmente, y puesto que la salida que genera un script PHP suele ser HTML, se suele emplear una cadena de texto convenientemente formateada en HTML, o bien una función auxiliar que formatee un mensaje de error arbitrario, que se le pase como argumento.

Esta visión de la gestión de errores puede ser válida en ocasiones, o para pequeñas aplicaciones, pero en general presenta los siguientes problemas:

- No se pueden generar errores en cualquier parte de la aplicación, por ejemplo en una función auxiliar que se llame mientras se genera código HTML, ya que el mensaje saldría mezclándose con el código HTML anteriormente generado.
- Hay ocasiones en que es necesario terminar lo que se está haciendo antes de parar la ejecución del script (por ejemplo para no dejar inconsistencias en datos relacionados entre sí que han sido parcialmente modificados o generados).
- La aplicación no se puede recuperar del error fácilmente cuando éste se produce, excepto mediante complejas estructuras condicionales, que deben repetirse en cada momento. Esto se complica además si queremos establecer varios niveles de error.

Estas cuestiones se ven facilitadas si existe una gestión de errores eficaz en la aplicación. Por tanto hemos decidido que en nuestra aplicación existirá una gestión de errores, de forma que éstos se presenten al usuario de la manera más conveniente en cada momento (que no tiene por qué venir dada por el flujo de ejecución de la aplicación) y que además permita que la aplicación pueda seguir su ejecución aún habiéndose producido errores en la misma, en este caso recuperables (que igualmente deben notificarse al usuario).

Para ello, se ha utilizado una clase a su vez utilizada por patForms para gestionar sus propios errores y que podemos personalizar a nuestro gusto: en concreto es **patErrorManager** (que es parte del paquete patError). La descripción de patError, en palabras de su autor, es la siguiente:

> ¿QUÉ ES PATERROR?
>
> Inspirado por la gestión de errores existente en PEAR, patError intenta resolver el problema de gestionar los errores en tiempo de ejecución que se producen en el interior de una aplicación PHP. De esta forma patError proporciona al desarrollador una

interfaz sencilla para "lanzar" (*throw*) errores o devolver un objeto de error como un valor de retorno. Por otro lado, los errores pueden "cazarse" (*catch*) automáticamente mediante el manejador de errores que se haya registrado.

patError también soporta diferentes niveles de error. Los tres niveles bien conocidos de: notificación (*notice*), aviso (*warning*) y error (*error*) vienen incluídos y pueden utilizarse sin configuración previa. patError también proporciona un comportamiento predefinido por defecto para cada uno de estos tres niveles de error.

Además de estos tres niveles de error incluídos, se pueden registrar niveles de error personalizados en tiempo de ejecución. Cuando se pasa de un estadio de "desarrollo" o "pruebas" al de "producción", puede que se necesite cambiar también el manejo de los errores pasando de un modo "exhaustivo" a otro más "silencioso". Para estos propósitos, patError proporciona seis modos diferentes de manejo de errores:

- *ignore* (ignorar - no hace nada)
- *trigger* (disparar - lanza el correspondiente error interno de php)
- *echo* (hacer eco - imprime el nivel del error y su mensaje)
- *verbose* (exhaustivo - igual que *echo*, pero imprime además el campo de información del error)
- *callback* (llamada automática - llama a una función definida por el usuario)
- *die* (morir - termina la ejecución, con un mensaje)

El manejador *"callback"* es el más potente. Mientras que los demás métodos son funciones incluídas de serie en patError (o, para ser más exactos, en patErrorManager), el manejador *"callback"* lo que hace es llamar a una función registrada por el usuario para el tratamiento de los errores. De esta forma, puedes escribir tus propias funciones o clases de manejo de errores adecuadas para tu aplicación. Por ejemplo, se pueden utilizar manejadores de errores personalizados para escribir en ficheros de log o enviar e-mails.

...

Y continúa con la siguiente reflexión:

¿Por qué utilizar patError?

1) patError está orientado a objetos. El gestor de errores retorna objetos de error que pueden manejarse con facilidad.

2) API estática. patErrorManager soporta una interfaz estática para configurar y lanzar errores. De esta forma no hay que andar jugando con referencias a objetos en el interior de la aplicación.

3) Fácil de usar. Simplemente dí: "`patErrorManager::raiseError(123, 'Mi error personal', 'Alguna información adicional');`". La API al completo está constituída por diez funciones:

- Tres de ellas son para propósitos de configuración: setErrorClass, setErrorHandling, registerErrorLevel
- Dos funciones de utilidad: getErrorHandling, translateErrorLevel
- Cuatro métodos para lanzar errores: rase, raiseError, raiseWarning, raiseNotice
- Uno para comprobar valores de retorno: isError

4) Fácil de instalar e integrar. Simplemente necesitas copiar dos ficheros en el directorio de inclusiones, incluír uno y patErrorManager hará el resto. patError también se puede instalar mediante el instalador de PEAR.

5) patError es extensible. Te permite añadir tus propios manejadores de errores. Implementar un manejador es muy simple: sólo se necesita una función global o un objeto que dé soporte a la función de manejo de errores.

6) patError es gratis y abierto! Al igual que las otras pat-clases, patError es software libre y –por supuesto– viene distribuido en forma de código fuente.

Como se puede ver, este paquete (constituido por las clases **patError** y **patErrorManager**) proporciona una gestión de errores bastante flexible, sencilla y conveniente para nosotros.

Si bien la gestión de errores que ofrece PEAR, mediante su paquete `PEAR::Error`, es tan buena (o más) como *patError*, finalmente nos decidimos por esta última, por la sencilla razón de que facilita la gestión de los errores que puedan producir el resto de las clases del mismo autor utilizadas en la aplicación (esto es: *patForms*, *patTemplates*, y otras), ya que configurando la gestión de errores con patError de forma general en la aplicación, los errores que generen estas otras clases también vendrán gestionados de la misma manera, sin necesidad de cambiar nada más.

Descripción del manejo de los errores en la aplicación

Para poder presentar al usuario los errores en el momento que sea más conveniente, se ha decidido utilizar la siguiente técnica de gestión de errores:

- Cuando se genere un error, éste se almacenará en un almacén de errores global de la aplicación.
- En este punto, se puede decidir si seguir la ejecución normal del programa o bien, variar el flujo normal de ejecución (por ejemplo, retornando un objeto de error)
- En cualquier momento, se puede saber si se ha producido un error fatal que obligue a terminar lo que se estaba haciendo.
- Cuando sea posible, se presentarán los errores (o notificaciones) al usuario, que podrá ser junto con los datos resultantes de la operación solicitada.

Con ayuda de patError es relativamente sencillo implementar ésto. Se ha decidido utilizar una función callback que actuará de una forma u otra (almacenará el error, o lo presentará en pantalla) dependiendo del momento en que nos encontremos en la aplicación. Así, por ejemplo, durante todo el programa se estará en "*modo almacenar*" excepto en el momento de presentar al usuario el formulario con los datos leídos o procesados, en el que se presentarán, además, aquellos errores o notificaciones que se hayan podido producir durante este proceso.

En la sección 9.4.4 se puede ver la forma concreta como se ha implementado ésto, mediante la clase estática **ManejoErrores**, que contiene el método callback mencionado, así como otros métodos de apoyo.

9.4. Detalles de la implementación

Aquí veremos de una forma más exhaustiva detalles de la implementación realizada en casos de especial interés, por los problemas que constituyeron y cómo se han ido resolviendo.

9.4.1. Clase "Configurador" – Modos de operación

(Sobre el atributo "modo" de la clase "Configurador")

Como se ha explicado anteriormente, el hecho de instanciar el objeto en “MODO_REFRESCAR” varía el funcionamiento interno de la clase, y concretamente implicará que no se debe procesar órden alguna procedente de la interfaz de usuario, sino únicamente leer los ficheros de configuración (y, especialmente, los ficheros de resúmen) para calcular los nuevos parámetros afectados y escribir los cambios. El “MODO_REFRESCAR” pues se trata pues de un modo de operación limitado o reducido para el módulo.

Para instanciar el objeto en uno u otro modo, y poder identificar dicho modo sin equivocación posible, se han definido dos constantes en el fichero “includes/clase_Configurador.php”, que son:

FW_MODO_NORMAL (con valor 0)

FW_MODO_REFRESCAR (con valor 1)

El constructor de la clase padre “Configurador” asigna el valor al atributo 'modo' según se le haya pasado como parámetro (siendo por defecto FW_MODO_NORMAL).

En la implementación, hay que tener en cuenta que una clase en “MODO_REFRESCAR” no debe procesar ningún *Submit*, ni obtener valores de los formularios, ni tampoco presentar interfaz gráfica alguna. Su única función será volver a cargar la configuración, con los datos auxiliares provenientes de los ficheros de resúmen (que serán los que más probablemente hayan cambiado, si se le ha invocado en este modo), y re-escribir dicha configuación para, de esta forma, hacerla efectiva.

9.4.2. Control de los Servicios del sistema

(Acerca de los métodos de control de servicios del sistema de la clase “Configurador” y sus subclases)

Para que un módulo pueda controlar la ejecución del servicio del sistema íntimamente relacionado con el mismo (por ejemplo: el módulo del *Proxy Squid* necesitará controlar la ejecución del servicio del sistema squid), se han creado una serie de métodos en la clase “Configurador” que son:

- servicio_en_ejecucion()
- ejecutar_servicio($accion)
- servicio_activo_arranque()
- configurar_servicio($accion)

Mientras que los dos primeros tratan de controlar la ejecución *inmediata* del servicio (es decir, el estado en que se encuentra el demonio en concreto en este mismo instante), los dos últimos se emplean para modificar la configuración de inicio del servicio (es decir, si el sistema operativo lanzará el servicio en el momento del arranque).

El parámetro $accion es la acción que se desea ejecutar o configurar, y puede tomar un valor de los siguientes: (definidos en 'includes/clase_Configurador.php')

FW_SERVICIO_PARAR (0)

FW_SERVICIO_INICIAR (1)

FW_SERVICIO_RECARGAR (2)

Así, se indica si se desea iniciar o parar un servicio (o incluso si se desea *recargar*, esto es, lanzar una señal al demonio para que vuelva a leer sus ficheros de configuración y así adquiera los cambios de forma inmediata sin detener su ejecución).[3]

[3]En el caso de emplear estas constantes como argumento del método configurar_servicio(), el significado será: FW_SERVICIO_INICIAR para configurar el inicio automático del servicio en el arranque; FW_SERVICIO_PARAR

9.4.3. Implementación de la definición de formularios en XML

(Sobre la clase "patForms_ Definition", empleada a la hora de cargar los formularios de "patForms" a partir de descripciones XML)

Esta clase se ha re-escrito a partir de la original `patforms_Definition` encontrada en la distribución de patForms, ya que sólo era compatible con PHP versión 5. Para hacerla compatible con PHP 4 (la versión de PHP que utilizamos es nuestra distribución Debian) se ha modificado el código de inicialización de la clase (constructor), la declaración de variables, y la especificación de la visibilidad de éstas así como de los métodos.

También se ha corregido un fallo de la clase original acerca del manejo de los campos de tipo Enum (así es como se llaman en patForms a los SELECT de HTML). Este fallo consistía en que, si bien al pasar de Array a la representación en XML los índices de tipo numérico generaban un nuevo tag (que por defecto se le ha dado el nombre `<tag>`), en la transformación inversa estos elementos se conservaban con dicho nombre ('tag') cuando lo correcto sería ignorar dichos elementos; haciendo ésto, los índices numéricos se regeneran automáticamente. Esta corrección se puede ver en la implementación del método **read()**.

El código de la clase modificada se presenta en el anexo I en la página 163 de esta memoria.

9.4.4. Clase estática ManejoErrores - Gestión de errores de la aplicación

Como se comentó anteriormente, en nuestra aplicación se ha realizado una gestión de errores basada en la clase patErrorManager (que es parte del paquete patError), que proporciona una gran flexibilidad a la hora de realizar el manejo de las condiciones de error.

En concreto este manejo se ha realizado en nuestro caso mediante la llamada a una función *callback*, esto es, una función a la que el sistema efectúa una llamada de forma automática en cualquier momento en que se produzca una condición de error.

Nuestra técnica de gestión de errores (explicada en la sección 9.3.3) consiste en que dicha función *callback* almacene este error y devuelva el control, de forma que la aplicación pueda seguir su curso (si así se desea) o bien detectar esta situación de error y controlarla de una forma más específica en ese momento (mediante el código condicional habitual). Posteriormente, cuando llegue el momento adecuado, se presentará este error al usuario (junto con otros errores y notificaciones que se hayan podido también producir) convenientemente formateado.

Para ello, se ha implementado la clase estática ManejoErrores, cuya razón principal de existencia es la de realizar el manejo de las condiciones de error que se produzcan, de forma generalizada en la aplicación. Esta clase se ha ideado como una clase *estática* por las siguientes razones:

- Sólo existirá una gestión de errores global en la aplicación, que afectará por igual a todos sus módulos. Por tanto no va a ser necesario tener varias instancias de esta clase en nuestra aplicación.
- Para no tener que mantener referencias al objeto en todos aquellos casos en que necesitemos llamar a sus métodos, lo cual complicaría innecesariamente la gestión de errores. Simplemente, en el momento en que sea necesario, anteponiendo el nombre de la clase (ManejoErrores::) se consigue llamar al método deseado.

Dicha clase contendrá un método de especial importancia que es el que realiza el manejo del error (en concreto es el método `maneja_error()`) que es el método que se configurará como

para desactivar el servicio en el arranque; `FW_SERVICIO_RECARGAR` no debe usarse pues no tiene sentido en este contexto.

manejador *'callback'* en patErrorManager. Dicho método actuará de la forma deseada en cada momento (almacenando el error, o presentándolo por pantalla) según el modo de funcionamiento que hayamos configurado en la clase. Cuando se produzca una condición de error (lanzada mediante los métodos raise() de patErrorManager), esta última clase llamará al método maneja_error(), pasándole como argumento los datos del error.

Como ejemplo de cómo se realiza la notificación de un error a nivel de código, veamos una porción de código real de la aplicación:

```
clase_Configurador_Clamav.php - KWrite
Archivo  Editar  Ver  Marcadores  Herramientas  Preferencias  Ayuda

    }

    /**
      * Realiza las acciones 'interactivas' que solicite el usuario (botón Actualizar)
      * (Nota: No se llamará a este método en el caso de Refrescar Módulo)
      */
    function ejecutar_acciones() {
        //comprobamos si hay que actualizar la bd de virus
        if(isset( $_POST['Actualizar'] )) {
                exec($GLOBALS['fw_config_sudo']." freshclam --stdout --quiet", $salida, $retorno);
                if ($retorno == 0 || $retorno == 1) {
                    if (!empty($salida))
                        patErrorManager::raiseWarning(FW_WARNING_SALIDA_COMANDO,
                        nl2br(htmlspecialchars(implode("\n",$salida))));
                    if ($retorno == 1)
                        patErrorManager::raiseNotice(FW_NOTICE_ACTUALIZACION_FRESHCLAM_EXITO, "La base
                        de datos de virus ya está al día. No es necesario actualizar.");
                    else
                        patErrorManager::raiseNotice(FW_NOTICE_ACTUALIZACION_FRESHCLAM_EXITO,
                        "Actualización realizada con éxito.");
                } else {
                    if (!empty($salida))
                        patErrorManager::raiseError(FW_ERROR_ACTUALIZACION_FRESHCLAM,
                        nl2br(htmlspecialchars(implode("\n",$salida))));
                    else
                        patErrorManager::raiseError(FW_ERROR_ACTUALIZACION_FRESHCLAM,
                                "Hubo un error en la actualización de ClamAV.<br>\n".
                                "Código de salida de <tt>freshclam</tt>: $retorno");
                }
        }
    }

    /**
      * Lee ficheros XML de definición del formulario
```

Figura 12: Ejemplo de código de gestión de errores

En este caso, se ve claramente las posibilidades que tenemos. En el código mostrado, se ejecuta un comando del sistema (en este caso la utilidad `freshclam`, para actualizar la base de datos de virus del antivirus ClamAV) y dependiendo del código de salida del mismo, se presenta al usuario el mensaje información o error correspondiente.

Además, tenemos dos posibilidades a la hora de lanzar una condición de error. Por ejemplo:

```
function procesar() {
    ...
    if (<condicion de error>) {
        patErrorManager::raiseWarning(101, "Atención, se ha producido un error");
    }
    ...
}
```

Este sería el caso de un error leve (en este caso, un *aviso*). Se produce una condición de error (que si nos encontramos en "modo almacenar", se notificará al usuario más tarde) pero la ejecución de la función sigue su curso. Mediante una construcción condicional (`if–else`) se puede variar la forma en que actúa la función, pero la ejecución de ésta no se detendrá y finalmente realizará su procesamiento y retornará algún resultado. Otra posibilidad es:

```
function procesar() {
    ...
    if (<condicion de error>) {
        return patErrorManager::raiseError(201, "Error: no se puede completar el proceso");
    }
    ...
}
```

Este sin embargo es un error más grave. La función en curso detiene su procesamiento normal y retorna un objeto de error (el objeto patError que se construye con la información proporcionada). El llamante podrá comprobar si el resultado de la función generó un error (mediante patErrorManager::isError()) y decidirá si seguir con el procesamiento, recuperarse de la condición de error deshaciendo acciones anteriores, o bien sencillamente no hacer nada y dejar que el mensaje de error se muestre al usuario como si de una notificación se tratase (junto con el resto de los datos producto del procesamiento hasta el punto al que se ha podido llegar). Esto queda a decisión del programador, y proporciona suficiente flexibilidad allá donde se necesite.

En el anexo J en la página 168 se describe la interfaz de la clase estática ManejoErrores con detalle.

9.4.5. Procesamiento de ficheros. Clase Config y Extensiones

Durante la ejecución de la aplicación se accede constantemente a diversos archivos: archivos de configuración (escritos siguiendo formatos heterogéneos), archivos de definición de estructuras de datos (escritos en XML), archivos de configuración de la propia aplicación y otros. Estos archivos se procesan en busca de parámetros y otros datos, se modifican, se crean, se eliminan..., en definitiva era preciso disponer de un conjunto de funciones capaces de gestionar ficheros de texto con formatos de lo más diverso.

Como ya se comentó en secciones anteriores de esto se encargan los métodos de la clase Config, de la biblioteca PEAR:Config. Esta clase permite el procesamiento de archivos de texto en los siguientes formatos estándares y conocidos:

- Ficheros de configuración de Apache
- Ficheros Ini.
- Ficheros Ini con comentarios.
- Ficheros XML.
- Ficheros PHP con definiciones de arrays.
- Ficheros PHP con definiciones de constantes.
- Ficheros de configuración genéricos.

En muchos de los casos estos tipos de ficheros, "contenedores" siguiendo la nomenclatura de Pear::Config, nos resultaron suficientes para nuestros propósitos. Sin embargo, en otras no se ajustaban al formato de los ficheros que requerían ser procesados y para solucionar este problema decidimos extender la clase Config añadiendo algunos tipos de archivos soportados más.

El código de estas extensiones puede encontrarse en el directorio "*configExtensions/Container/*", de la aplicación. Y consta de los siguientes archivos:

- dglists.php
- firehol.php

- snort.php

En cada uno de ellos se crea una clase específica con sus correspondientes variables de clase y los métodos:

- Constructor.
- parseDatasrc (adquiere los datos del fichero, organizándolos según secciones, comentarios, directivas, valores y nombre, etc).
- toString (crea cadenas de texto que mostrar o escribir en los ficheros a partir de parámetros y sus atributos).

A la hora de utilizar alguna de estas extensiones, la estructura de la biblioteca Pear::Config no nos deja otra opción que añadir un nuevo campo en el array de tipos de archivos soportados. Por tanto antes de usarlas (al inicio de los archivos de definición de las clase en las que se emplean) se incluye una linea como la que sigue (en este caso para el parseo de archivos de configuración de Snort):

> $GLOBALS['CONFIG_TYPES']['snort'] = array('./configExtensions/Container/snort.php', 'Config_Container_Snort');

9.4.6. Validación de campos de formulario. Extensión de patForms

Cuando una aplicación adquiere datos introducidos por el usuario es fundamental realizar algunas comprobaciones sobre estos. Una vez adquiridos estos datos serán utilizados en el código de la aplicación y las recomendaciones de programación segura exigen una validación previa.

En nuestro caso existe además una responsabilidad añadida, no podemos rellenar los archivos de configuración de los distintos módulos gestionados con datos erróneos o mal formados; ya que aun no recibiendo error alguno al ejecutar la aplicación web de configuración, podríamos producir fallas de funcionamiento o agujeros de seguridad en estos programas configurados por medio de la aplicación.

Por ejemplo no podemos permitir que un usuario introduzca caracteres alfabéticos en un campo donde se requiere un puerto o una dirección de red en formato numérico. Es preciso avisar al usuario e impedir la escritura de los datos en el fichero de configuración de destino.

PatForms implementa métodos que permiten la auto-validación de los formularios. Se asocian a los campos del formulario diversas reglas que han de cumplir los valores introducidos en dichos campos. Al recibir el "submit" del formulario se auto-comprueba que las reglas se han cumplido y en caso contrario se generan errores que procesaremos con patError de la manera que se comentó en la sección 9.3.3 en la página 65 y en 9.4.4 en la página 69.

Aunque patForms incluye múltiples reglas de validación, encontramos algunos casos particulares de nuestra aplicación no contemplados. Por tanto fue necesario extender la clase patForms_Rule de la biblioteca patForms. A continuación se enumeran los ficheros donde encontrar estas nuevas reglas de validación. Estos ficheros están disponibles en el directorio "*patFormsExtensiones/*" de la aplicación.

- GroupConditionalRequired.php. Esta regla puede ser usada para hacer que sea obligatorio introducir datos en algunos elementos, si se introducen datos en uno solo de un grupo de elementos en el formulario.
- Ipaddress.php. Se comprueba que la sintaxis de una dirección de red IP es correcta.

- Ipport.php. Se comprueba que los valores introducidos corresponden con los de un puerto tcp.
- Number.php. Se comprueba que los valores introducidos en el campo son números.

10. Creación de distribución en CD auto-arrancable

Una vez realizado el estudio de las distintas herramientas de seguridad, de sus configuraciones y modos de funcionamiento. Y tras la integración de estas con el objetivo de hacerlas trabajar conjuntamente y complementándose, desarrollamos una aplicación web capaz de simplificar la configuración de cada uno de los módulos estudiados, facilitando al usuario no avanzado una básica configuración del sistema, segura y fiable.

Un paso más en nuestro intento de simplificación, es ofrecer al usuario una distribución completamente adaptada a sus necesidades. Desde nuestro punto de vista Debian/GNU Linux, es uno de los más completos sistemas operativos, dispone de paquetes de todo tipo y soporta casi cualquier arquitectura. Sin embargo, debido a esta diversidad, un usuario novato puede encontrarse perdido en este mar de paquetes desconocidos.

Nuestro propósito es seleccionar el subconjunto de paquetes necesarios en una maquina que realice las funciones de cortafuegos, o servidor de seguridad perimetral, además de los paquetes básicos que todo sistema debe tener, y algunos paquetes adicionales de utilidad. Con todos esos paquetes realizaremos un CD auto-arrancable, *Live-CD*, de manera que el usuario no tenga que instalar el sistema en el disco duro.

De esta manera, el usuario podrá probar las herramientas integradas y la interfaz de gestión y configuración desarrollada, sin necesidad de instalar todo el sistema en su equipo.

10.1. Proyecto Metadistros

Metadistros es un sistema y una infraestructura para crear distribuciones a la medida de grupos concretos de usuarios (universidades, colegios, empresas, cursos itinerantes bajo Linux...), por tanto es exactamente lo que necesitamos.

Empezaremos aclarando que se entiende por distribución, una distribución es el conjunto formado por:

Sistema Operativo, entendiendo por tal al programa que gestiona todos los recursos del equipo y los pone a disposición del usuario. En un sistema tipo UNIX el sistema operativo está formado a su vez por:

- Un núcleo, que en el caso de las distribuciones GNU/Linux, será Linux y en otros Sistemas Operativos libres puede ser un núcleo BSD, Hurd, Mach u otro. Es el que media entre las aplicaciones que interaccionan con el usuario y la gestión de los recursos hardware del equipo.
- Un conjunto de utilidades básicas que permiten interaccionar al núcleo con el usuario como el shell, los editores, etc, que en el caso de las distribuciones GNU/Linux son en una gran parte las utilidades del Proyecto GNU y otras provenientes de otros proyectos como el BSD. Una contribución muy importante a que el núcleo Linux sea soportado en tantas plataformas hardware es que GCC las soporta. Es por todo los anterior y en especial lo referente a GCC, por lo que se denomina Sistema Operativo GNU/Linux y no Linux.

Aplicaciones

- Escritorio.
- Utilidades de configuración/actualización.
- Utilidades de comunicación (Navegador, Correo, Chat).

- Entornos de desarrollo.
- Servicios de red.
- Otras.

El proyecto Metadistros está patrocinado por Hispalinux, la oficina del software libre de la ULPGC, y software-libre.org. Podemos encontrar información detallada del proyecto en su sitio web, `http://metadistros.software-libre.org`. Según leemos en los anteriores enlaces el objetivo último del proyecto, consiste en "la creación de una infraestructura para que cualquiera pueda crear una distribución a medida".

MetaDistros es un proyecto cuya finalidad principal es la de proporcionar a la comunidad de software libre toda una infraestructura para la creación sencilla de distribuciones de GNU/Linux a medida. Utilizando el sistema de MetaDistros, es posible crear un CD desde el que arrancar y utilizar esta distribución.

Existe la posibilidad de lanzar el arranque en el llamado modo "live", en el cual el sistema completo arranca y se ejecuta sin necesidad de utilizar el disco duro. El arranque en modo "live" es realmente interesante porque permite probar el sistema sin interferir con cualquier otro sistema operativo que pudiera estar instalado en el ordenador del usuario. Además, se da la posibilidad de instalar esta distribución en el disco duro una vez probada. De esta manera, el resultado es que se crea una distribución a medida para cumplir unos determinados objetivos.

El usuario de esta distribución la arranca en "live" para comprobar si es de su agrado. Si éste es el caso, procede a la instalación en el disco duro. MetaDistros provee de la infraestructura necesaria para conseguir estos objetivos.

Este sistema consiste en 2 partes fundamentales:

La Distribución, es una distribución GNU/Linux ya instalada y configurada para el uso elegido. Ésta va dentro del CD-Rom junto con el Calzador.

El Calzador, encargado de que la distribución funcione desde el CD-Rom y de instalarla al disco duro. Es independiente de la distribución que se use.

El sistema de MetaDistros realmente se ocupa del desarrollo y puesta a punto del calzador y de las utilidades. Será el usuario final el que se encargue de crear, adaptar y configurar a medida una distribución de GNU/Linux cualquiera, a partir de la cual, utilizando el sistema de MetaDistros, se generará un CD auto-arrancable que permitirá la ejecución en "Live" y la instalación en el disco duro de la misma.

10.1.1. La distribución base

En la figura 13 en la página siguiente, se representa la infraestructura que provee el sistema Metadistros.

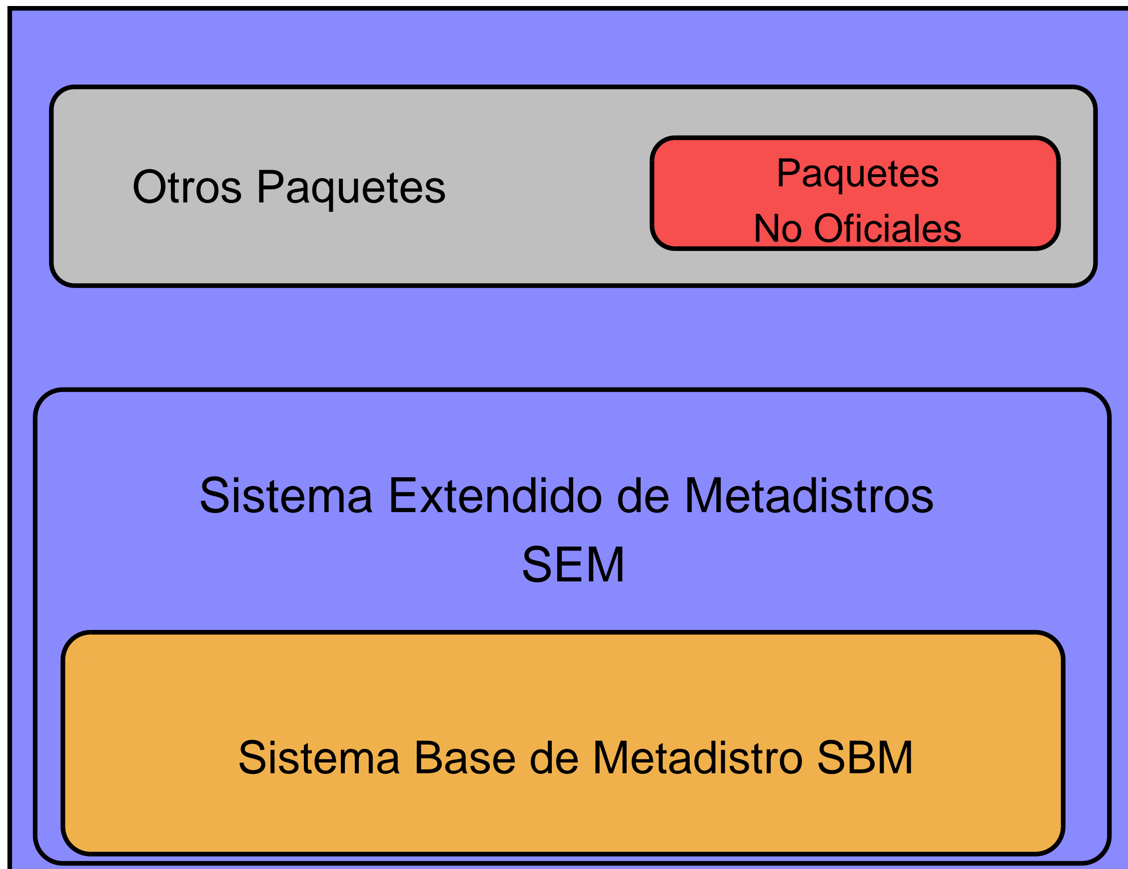

Figura 13: Infraestructura del Sistema Metadistros

Tanto el sistema base de metadistros (*S.B.M.* a partir de ahora) como el sistema extendido de metadistros (*S.E.M.*) forman parte de la infraestructura que provee MetaDistros. El objetivo, tanto del *S.B.M.* como del *S.E.M*, es proporcionar al usuario final una distribución de partida para que la personalice a su gusto. De esta manera, se facilitan las cosas y se produce un ahorro de tiempo para el usuario de MetaDistros.

A continuación, se describirán ambos sistemas, base y extendido:

S.B.M: Consiste en un sistema GNU/Linux básico, con el menor número de aplicaciones posibles para que resulte completamente funcional a unos niveles básicos. Para ello, incorpora aquellas utilidades y servicios mínimos sin los cuales el sistema no es mínimamente funcional. En la práctica, este S.B.M. coincide con el Sistema Base de Debian.

S.E.M: Este sistema incluye al S.B.M. y además se incorpora otro conjunto de paquetes que proporcionan nuevas funcionalidades a la distribución base: internacionalización, configuraciones por defecto y algunos otros paquetes adicionales. La idea es aumentar la funcionalidad del S.B.M. sin entrar en una personalización profunda del sistema.

Es importante señalar que tanto el S.B.M. como el S.E.M. son herramientas que MetaDistros pone a disposición del usuario final para facilitar la generación de una meta-distribución, pero su uso no es obligatorio. En realidad, el único requisito para conseguir una meta-distribución a medida es partir de alguna distribución de GNU/Linux funcional y adaptada a las necesidades del usuario. Ésta puede generarse de muchas formas diferentes. Por ejemplo, se puede partir de una

distribución como Red Hat, Mandrake o Debian. O bien, es posible partir de una meta-distribución ya generada. En ambos casos, habrá que realizar una personalización de la misma. El resultado es lo que se denomina distribución base. Hecho lo anterior, el siguiente paso es utilizar el resto de herramientas de MetaDistros para la generación de la meta-distribución a partir de la distribución base.

10.1.2. El calzador

El calzador constituye una de las partes principales de este proyecto. Su misión es "calzar" una distribución de GNU/Linux cualquiera (en la práctica este objetivo no ha sido totalmente conseguido, siendo únicamente las distribuciones basadas en Debian las totalmente soportadas), en un CD-ROM; de manera que se permita su posterior ejecución e instalación en el disco duro del usuario.

El concepto de "calzar" se puede entender como el proceso por el que una distribución GNU/Linux es adaptada y configurada de forma que es perfectamente utilizable desde un CD-ROM de manera análoga a como se utilizaría si estuviera instalada en el disco duro local. Esto es lo que se conoce como ejecución en "live" (o en "vivo").

Para que esto sea posible, es necesario realizar una ingente cantidad de tareas diferentes, que son llevadas a la práctica por el calzador. Dentro de las funciones principales del calzador, se encuentran las siguientes:

Arranque del CD-ROM: El calzador es responsable de que el sistema arranque de manera adecuada desde un CD-ROM, en lugar de que se produzca desde el disco duro o de alguna otra forma alternativa. Para ello, debe ser capaz de arrancar desde distintos tipos de dispositivos, siendo los más típicos los de tipo IDE o SCSI.

Compresión/Descompresión de la distribución base: Para optimizar el espacio ocupado por la distribución base y maximizar la cantidad de software y de programas incluidos en el CD-ROM, se utilizan técnicas de compresión a la hora de almacenarla la imagen de la distribución en el CD-ROM. Serán necesarias tareas de descompresión de los datos durante la ejecución del CD. Todo esto es llevado a cabo por el calzador.

Detección de los dispositivos (hardware): Uno de los puntos críticos del calzador es la detección de hardware. El calzador debe ser capaz, durante el arranque del CD-ROM, de detectar el hardware instalado en el equipo y proceder a cargar los módulos necesarios del núcleo para darles soporte y poder utilizar los distintos dispositivos. Cuando se produce la instalación en el disco duro, la configuración de los distintos dispositivos se mantiene, de manera que pueden seguir utilizándose sin ningún problema.

Configuración básica del sistema: Otro apartado muy importante del calzador es la configuración del sistema que lleva a cabo. Durante el arranque, tiene lugar la creación de diversos ficheros de configuración críticos a partir de plantillas y en función de los dispositivos detectados, como pueden ser los ficheros /etc/fstab o / etc/X11/XF86Config- 4.

Instalación en el disco duro: Es igualmente importante el instalador. Como su propio nombre indica, su misión es realizar la instalación de la distribución que se está ejecutando en "vivo" (live), en el disco duro, manteniendo la configuración básica del sistema y la configuración del hardware del equipo. Durante la instalación, se preguntará al usuario por algunas opciones de configuración, como puede ser la creación de algún usuario para el sistema o la configuración de red del equipo.

En la figura 14 en la página siguiente, podemos apreciar esquemáticamente la estructura funcional del calzador.

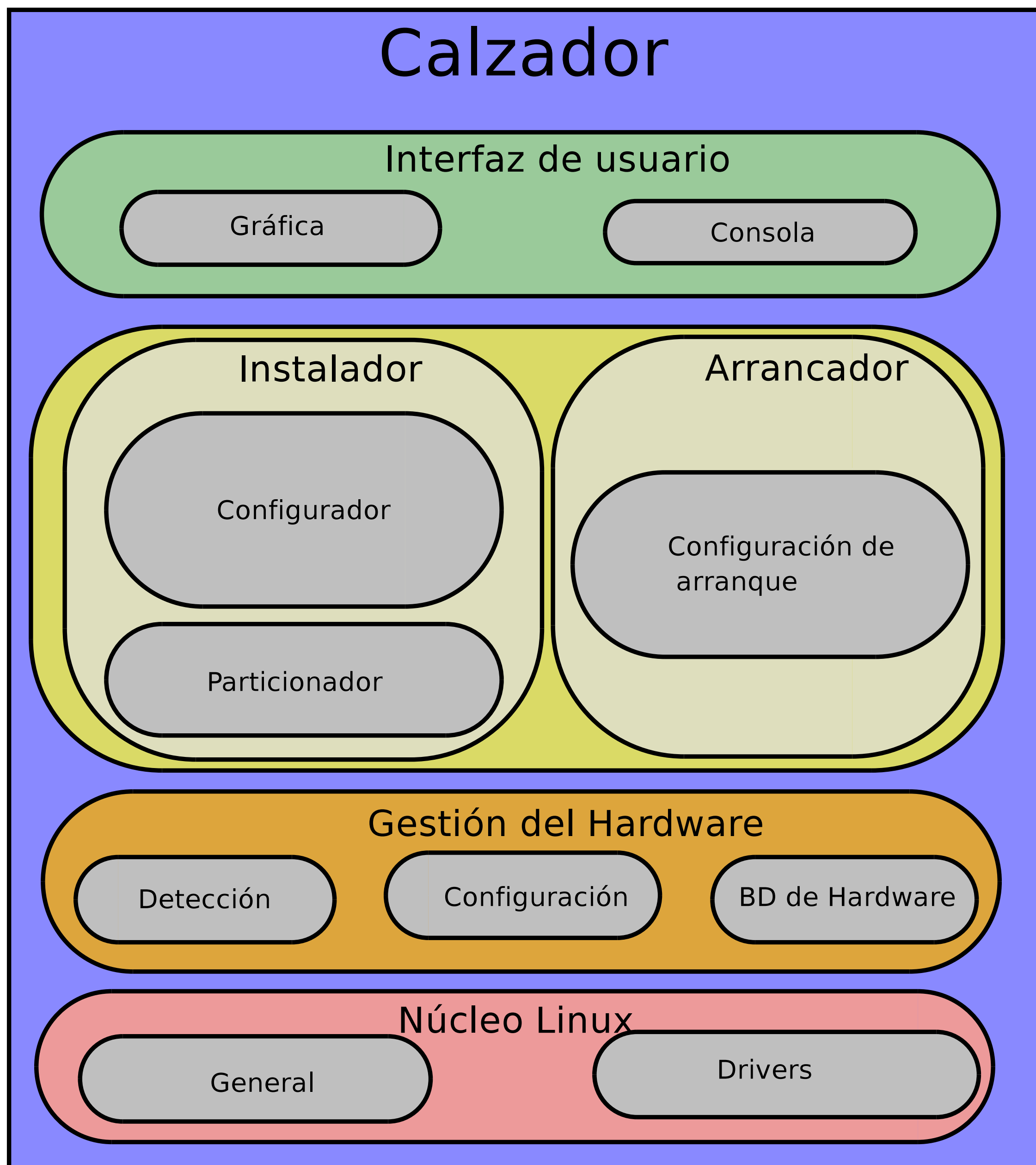

Figura 14: Esquema funcional del calzador Metadistros

10.2. Arranque del sistema

Antes de explicar en detalle el funcionamiento de Metadistros y sus componentes, es conveniente entender el proceso de arranque de un sistema GNU/Linux. Esta explicación se encuentra detallada en profundidad en los anexos de esta memoria de proyecto (anexo K en la página 171).

En el caso de Metadistros, el proceso de arranque tiene lugar desde un CD-ROM. Para crear este CD-ROM de arranque se hace uso de la herramienta ISOLINUX, que es capaz de arrancar el sistema desde un CD-ROM y pasar posteriormente el control al proceso "init".

En el anexo L en la página 173, se analiza en detalle el proceso que tiene lugar desde que la BIOS

pasa el control al CD-ROM hasta que pasa a ejecutarse el proceso "init". Todo este proceso se lleva a cabo a partir del conjunto de herramientas denominadas “calzador”.

10.3. Creación de Metadistro Anubix

En esta sección describimos los pasos seguidos para la creación de nuestra meta-distribución Anubix. Además de explicar el proceso de creación incluimos detalles de las adaptaciones del calzador y la distribución usada como base, (scripts modificados, imágenes de arranque, software instalado y configuración...).

Empezamos por comentar los requerimientos para el desarrollo de la “meta-distribución”:

- Un sistema Linux con las herramientas típicas. En nuestro caso Debian Sarge.
- Espacio suficiente en el disco duro para la distribución base, calzador y las imágenes ISO que se generarán.
- "mkisofs". Esta herramienta es fundamental para generar la imágenes del sistema y el CD arrancable.
- Memoria RAM o swap, o ambas cosas. Será necesario disponer de bastante memoria o en su defecto de bastante espacio swap (ya sea en partición o en archivo).
- Squashfs-tools, es una herramienta para crear y trabajar con sistemas de ficheros *squashfs,* que es el sistema que usaremos para incluir nuestro sistema base en el CD. Squashfs es un sistema de ficheros de solo lectura con alto grado de compresión; usa zlib tanto para comprimir ficheros, inodos y directorios. Para instlarlo usamos la herramienta apt, (*apt-get install squashfs-tools*).
- Squashfs-kernel-patch, además de las herramientas de squashfs, nuestro kernel de desarrollo debe soportar el sistema de ficheros. Es necesario por tanto descargar el parche adecuado (*apt-get install kernel-patch-squashfs)*, parchear las fuentes del núcleo de Linux y recompilarlo.
- Cloop. Si se quiere comprimir el sistema porque no cabe en el CD, se puede utilizar Cloop, que es un sistema de compresión de archivos loopback. Para ello será necesario instalarse las "cloop-utils". No es nuestro caso, ya que nuestro sistema sin comprimir no superará los 200MB.
- Alguna herramienta para grabar CD-Roms (cdrecord, etc).
- Qemu, es un emulador de varios procesadores que nos permitirá probar las imágenes que generemos sin tener que grabar las en CD.

Definiremos ahora la estructura de los directorios de trabajo que usamos en el desarrollo:

- **/mnt/md/sources** ->Es el directorio donde está el sistema que queremos usar en la “*distro*”. El sistema base de metadistros modificado según nuestras necesidades.
- **/mnt/md/master** ->Es el directorio en el que se guarda el contenido del CD. Éste contiene dos subdirectorios:
  - isolinux ->Aquí está el Calzador.
  - META ->Aquí está la distribución. Serán dos archivos del tipo “*Squashfs*” que generamos a partir del contenido del directorio */mnt/md/sources.*
- **/mnt/md/iso** ->Aquí se guardan las distintas imágenes ISO de la *distro.*

Una vez creados estos directorios, se procede a personalizar tanto el calzador como la distribución que usamos de base en el desarrollo. Estos pasos se describen en profundidad en 10.3.1 y en 10.3.2 en la página siguiente.

Proseguimos con la explicación del proceso de desarrollo, asumiendo que ya se han llevado a cabo las adaptaciones necesarias, indicadas en los puntos anteriormente citados de este documento.

El siguiente paso consiste en crear un archivo comprimido con *Squashfs* del directorio "/sources" donde está nuestra distribución. Este archivo al que llamaremos *META.squashfs*, debe quedar en el directorio *master/META/META.squashfs*.

A continuación creamos la imagen ISO con el contenido del directorio *master/* y el sector de arranque con el contenido de *isolinux*. Después de esto debemos tener la imagen de nuestra distribución en el directorio */mnt/md/iso*, totalmente preparada para ser grabada en un CD.

Antes de conseguir un resultado óptimo, fue preciso realizar múltiples pruebas. Para evitar tener que grabar en un CD en cada tentativa, usamos el emulador *qemu,* que nos permite ejecutar la distribución generada en una máquina virtual.

El anterior proceso ha de repetirse después de cada modificación o ajuste del calzador o del sistema base. Con el fin de automatizar el proceso en la medida de lo posible hacemos uso del script *creardistro.sh* cuyo código mostramos en el anexo M en la página 182.

10.3.1. Preparación del Calzador

En este punto nos centraremos en describir los pasos necesarios para adaptar el calzador estable de Metadistros a nuestra medida.

Para llevar a cabo esta personalización nos centraremos en los ficheros contenidos en el directorio /mnt/md/master/ (a partir de ahora para simplificar llamaremos al path completo $MASTER):

$MASTER/isolinux/conf/var.conf: Aquí configuraremos varios aspectos como el nombre de la distro (Anubix), idioma, si inicia con las X (en nuestro caso no), con DHCP, nombre de usuario y contraseña etc... (como nombre de usuario NUNCA debemos de elegir uno que ya exista en el sistema base, así si por ejemplo en el sistema base utilizábamos al usuario juan para personalizar el sistema base, aquí no podemos poner juan sino pepe, o paco o cualquier otro que no sea juan, pues como ya he dicho antes, el calzador crea un nuevo usuario y si se encuentra uno ya creado con el mismo nombre dará problemas)

$MASTER/isolinux/conf/q.conf: Aquí se configuran cuales van a ser las preguntas que se le harán al usuario al arrancar (nombre del usuario, clave del root, configuración de la red...)

$MASTER/isolinux/greeting: Lo que aparecerá en el mensaje de bienvenida. No se debe borrar o modificar la primera línea que es la pantalla de boot o inicio.

Con el fin de personalizar algo más nuestra distribución decidimos llevar a cabo algunas modificaciones estéticas, que aunque no son necesarias desde el punto de vista práctico darán una imagen propia a nuestro trabajo.

Para la edición de todos los gráficos necesarios para estas tareas que se describen a continuación se uso la potente herramienta libre The Gimp.

Una de estas personalizaciones es el cambio del bootsplash que metadistros trae fijado por omisión. Necesitaremos el paquete bootsplash descargable desde el repositorio de mentors (mentors.debian.net) que contiene la utilidad splash instable con el paquete bootsplash, además de algún tema que usamos como base para desarrollar el nuestro propio.

Lo primero es crear un nuevo archivo initrd, en nuestro caso de 10 megas de tamaño y con 7000 i-nodos en el que se incluirá las nuevas imágenes de bootsplash con el comando *splash.*

Finalmente debemos personalizar gfxboot, este es un gestor de arranque desarrollado por SUSE. Nos permite arrancar el sistema de distinta forma. Según la opción que eliga el usuario se pasarán unos parámetros u otros.

Además de encargarse de mostrar el menú de arranque y la ayuda en linea de dicho menú. Permite configurar el fondo de pantalla del menú y una imagen o animación al arrancar el sistema.

10.3.2. Preparación de la Distribución base. Sources

Una vez personalizado el calzador, archivos de configuración, scripts, imágenes, etc, conforme a nuestras necesidades, debemos personalizar la distribución base que usaremos. En principio el proyecto metadistros tiene como objetivo que esta distribución sea completamente independiente. Por lo tanto esta base podría ser cualquier distribución GNU/Linux, incluso un sistema creado desde cero por nosotros: *Linux From Scratch (LFS)*. Sin embargo, este objetivo no esta del todo logrado y el proyecto se orienta claramente a trabajar con una base Debian. Esto no solo no resulta un impedimento para nosotros, sino una ventaja, ya que desde el primer momento basamos nuestro desarrollo en Debian/Sarge.

Las posibilidades que encontramos, en cuanto a la preparación de la distribución base, para realizar nuestra meta-distro son cuatro:

Instalarse una distribución Debian normal en una partición del disco duro. Lo único que hace falta para hacer esto es una partición libre de aproximadamente 2 Gb (depende del tamaño que queramos para la distribución). Se coge el sistema que prefiera para instalar (Cds, disquetes, red, etc) e instala lo que se desee que haya en la distribución. No es necesario instalar nada en particular para que el sistema funcione, solo lo que se requiera o necesite. Después montamos la (o las) particiones donde hemos instalado este sistema base en el directorio sources donde estemos desarrollando la meta-distribución y seguimos con el desarrollo.

Coger un sistema ya instalado (como por ejemplo el que usa normalmente en su ordenador)

Utilizar el Sistema Base de Metadistros(SBM).

Remasterizar una distribución que se base en este sistema, como es el caso de Silu, Necromantux, Guadalinex... Remasterizar quiere decir modificar una Meta-distribución ya hecha.

Nuestra decisión ha sido utilizar el SBM. Esta es a nuestro entender la opción más simple, el SBM no es más que un sistema base de Debian Sarge, con el software básico y sin X. A continuación y a lo largo de todo este apartado veremos como adaptarlo a nuestras necesidades, instalando el software que necesitemos y configurándolo.

Al arrancar nuestra distribución en modo *Live* o al instalarla en el disco duro, conservara la configuración que hayamos dejado establecida durante el desarrollo de la meta-distribución. Por lo tanto, tras instalar los paquetes que irán en nuestra distribución hemos de configurarlos de la manera más estándar posible, para que se ajusten a las necesidades del mayor número de usuarios.

Por supuesto una vez que arranque el sistema, o este sea instalado, el usuario puede modificar la configuración a su gusto. Pero nuestro objetivo es que al menos se arranque con una configuración totalmente funcional.

Proceso de personalización del SBM Lo primero que debemos hacer es descomprimir el archivo que contienen el sistema base dentro de un directorio, *sources*, que emplazamos en nuestro directorio de desarrollo de la meta-distribución (*/mnt/md*).

Después procedemos a instalar los paquetes que necesitamos. Para ello haremos uso de la herramienta *chroot*. CHROOT es una llamada al sistema en UNIX que permite configurar un directorio como "raíz" del sistema de ficheros para un proceso y sus hijos. En otras palabras, permite configurar el sistema de forma tal que se puedan lanzar procesos confinados dentro de un determinado directorio. Para ellos, dicho directorio será el "/" (la raíz). Cualquier fichero o directorio que esté fuera del CHROOT les quedará inaccesible.

Entonces, procedemos de la siguiente forma:

```
anubix:/mnt/md# chroot sources/
```

Para poder acceder a Internet y realizar algunas otras tareas hemos de montar proc:

```
anubix:/# mount -t proc /proc proc
```

Podemos comprobar que todo esta correctamente y que estamos en el entorno "enjaulado", listando el directorio home y comprobando que esta vacío ya que no hay usuarios creados.

Lo siguiente es configurar la red para poder conectarnos a los repositorios de Debian con el comando apt. Así que modificamos los archivos, */etc/network/interfaces y /etc/resolv.conf* con los parámetros de nuestra red. A continuación reiniciamos el servicio de red, (*/etc/init.d/networking restart*) y comprobamos que tenemos conexión con el exterior. Actualizaremos la base de datos de paquetes y reconfiguramos las locales.

Haciendo uso de la herramienta apt-get instalamos todos los paquetes que decidimos incluir en nuestra distribución. Además de los paquetes básicos instalamos los módulos que integramos y que ya comentamos en apartados anteriores: squid, scavr, clamav, dansguardian, iptables, firehol y snort.

Entre los paquetes software instalados, hemos de destacar también aquellos necesarios para servir la aplicación web desarrollada y documentada en los apartados anteriores de esta memoria: el servidor web lighttpd, el servidor ssh, el interprete de php así como las diversas bibliotecas php ya comentadas (pear config, pat Forms, pat Errors, pat Templates, pat Sessions...).

Además se instalarán todas las dependencias requeridas por los paquetes instalados, esta tarea se simplifica enormemente gracias a las utilidades de gestión de paquetes apt y pear.

Por último se instala en el sistema base nuestra aplicación web de configuración. Esta queda ubicada en el directorio /var/www/anubix. Es importante tener en cuenta los permisos de archivos y directorios con el fin de que la aplicación funcione correctamente sin comprometer la seguridad del sistema.

La aplicación de configuración necesita ejecutarse con permisos de super-usuario, ya que necesita arrancar y parar servicios, y modificar archivos de configuración del sistema.

Para conseguir esto debemos configurar el servidor web ligero lighttpd para que ejecute los scripts php haciendo uso del módulo fast-cgi. Además de mejorar el rendimiento, esto nos permite definir un binario para ejecutar los scripts que sea propiedad del superusuario y tenga el bit SUID activado para que se ejecute con los permisos del propietario del fichero (root).

En el momento en que se ejecute el sistema Anubix, (en modo Live-CD), los programas y servicios tomarán la configuración que tengan en el momento en que se empaquetó nuestra Metadistribución.

Por tanto después de instalar cada componente, se configuró de forma que todos los módulos funcionen en un entorno estándar. Por ejemplo: el fichero de configuración de FireHOL se adaptó para su correcto procesamiento por la interfaz web de configuración, el visor de alertas de snort (snortalog) se configuró para adquirir los datos del directorio donde snort los deposita, etc.

11. Validación

Una de las fases fundamentales del desarrollo de software es la de validación. En ella obtendremos datos con los que realimentar otras fases del desarrollo y es fundamental para conseguir unos mínimos de calidad en el producto final.

Probar la aplicación consiste en generar una serie de casos de prueba, ejecutar la aplicación contra estos casos de prueba y observar que el comportamiento de la aplicación es el correcto.

En la parte final del desarrollo de este proyecto llevamos a cabo algunas pruebas de validación que expondremos a continuación. En algunos casos el resultado fue positivo y en otros, como era de esperar, no se pasó la prueba. Como un proyecto de fin de carrera tiene una extensión temporal limitada y los medios con los que se cuentan, temporales y humanos, también son limitados no todos los problemas han podido ser corregidos.

11.1. Pruebas de interfaces y contenidos

En esta etapa se revisa la forma en que se despliegan las páginas del sitio de forma precisa y se comprueba que se cumplan los estándares mínimos exigidos.

En el anexo N.1 en la página 183 de esta memoria podemos ver la lista de comprobación aplicada y los resultados obtenidos.

11.2. Pruebas funcionales y de operación

En esta etapa de pruebas se realizan comprobaciones completas de las funcionalidades de la aplicación, en nuestro caso: formularios, modificaciones de los ficheros de configuración, correcta gestión de servicios, etc.

En el anexo N.2 en la página 183 de esta memoria podemos ver la lista de comprobación aplicada y los resultados obtenidos.

11.3. Pruebas de carga

La carga de trabajo se refiere a la capacidad máxima que tiene un servidor web (hardware y software), para atender a un conjunto de usuarios de manera simultanea. Por ello, las actividades de esta etapa tienen relación con comprobar, de manera anticipada, el funcionamiento que tendrá el servidor del Sitio Web cuando este en plena operación.

En nuestro caso, por la misma esencia de nuestra aplicación, este apartado carece de gran importancia. Tratándose de una aplicación de configuración del sistema no se ejecutará por más de un usuario simultáneamente.

11.4. Pruebas de rapidez de acceso

Un aspecto importante a probar en una aplicación es la rapidez de acceso. Es decir como de rápido puede el usuario encontrar una la información que busca o el lugar al que necesita acceder.

En el anexo N.3 en la página 184 de esta memoria podemos ver la lista de comprobación aplicada y los resultados obtenidos.

11.5. Pruebas de accesibilidad

Para comprobar que un Sitio Web cumple con las normas de accesibilidad, la iniciativa WAI (Web Accesibility Initiative) de la W3C (World Wide Web Consortium), propone la realización de pruebas recogidas en las Directrices o Pautas de Accesibilidad para el Contenido de la Web de la cual nuestro objetivo es cumplir con la normativa WAI-A y para ello se realizaron algunas de las pruebas necesarias.

11.6. Pruebas de usabilidad

El test de usuario es una de las pruebas más importantes a realizar a una aplicación. Es la manera más cercana de aproximarse al uso real de esta. Por motivos de tiempo este test no ha sido realizado por usuarios reales, ni por un número suficiente de usuarios. Por tanto los resultados pueden no ser completamente exactos.

En el anexo N.4 en la página 184 de esta memoria podemos ver la lista de comprobación aplicada y los resultados obtenidos.