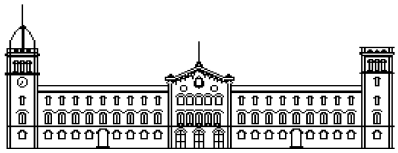


Treball fi de carrera

ENGINYERIA TÈCNICA EN INFORMÀTICA DE SISTEMES

Facultat de Matemàtiques
Universitat de Barcelona

Desarrollo de un dispositivo de filtrado de contenidos por rol de acceso.

Daniel Vidal de la Rubia

Directors: Jaume Timoneda Salat
Iván Sánchez Valencia
Realitzat a: Departament de Matemàtica Aplicada i Anàlisi. UB

Barcelona, 15 de junio de 2012

1 Resumen

El propósito de este proyecto es implementar el prototipo de un dispositivo de red que permita filtrar el acceso a Internet mediante listas en función del rol de cada usuario. Dicho dispositivo debe poder interceptar tanto las comunicaciones HTTP como las HTTPS, por lo que se deberá buscar una solución que implemente "Man In The Middle" para TLS/SSL.

Se buscará entre las soluciones de código abierto existentes en el mercado, para averiguar si el posible integrarlas dentro del dispositivo o es necesario implementar nuestras propias piezas de software.

2 Índice

3 Introducción

3.1 Motivación

Es normal que los padres recelen del juicio de sus hijos menores para evaluar si el contenido de determinados lugares de Internet es adecuado para ellos o no, lo que les lleva a querer controlar lo que sus hijos ven y experimentan en la red de redes, pero no siempre son capaces de ello (desconocimiento de cómo funcionan las tecnologías basadas en Internet, horarios incompatibles con la supervisión constante de sus hijos mientras navegan, etc.). Eso puede llevar a que los padres no permitan el acceso a Internet de sus hijos menores, aun siendo conscientes que un uso responsable de Internet es beneficioso para sus hijos dada la enorme cantidad de información que pueden acceder en la época de sus vidas que mayor capacidad de aprendizaje tienen y que "desconectarlos" de la red puede ser tan perjudicial para los niños como dejarlos que naveguen con total libertad por Internet.

Es por ello que muchos padres recurren al mercado en busca de algún tipo de solución que sea capaz de monitorizar las actividades de sus hijos en la red y notificarles el acceso a páginas de contenido "dudoso" para sus hijos o, incluso, bloquearles el acceso en casos especialmente peligrosos.

Y eso no es aplicable a familias solamente, también puede haber algunas instituciones como, por ejemplo, colegios e institutos que quieran proteger a sus miembros de los peligros de Internet mientras acceden libremente a los recursos beneficiosos de la red.

Por ello surge la idea de construir un dispositivo que sea capaz de filtrar los contenidos de forma inteligente, que sea capaz de entender el contenido de la página y decidir si es adecuada o no para el usuario en función de su perfil.

3.2 Diseño de la solución y funcionamiento básico

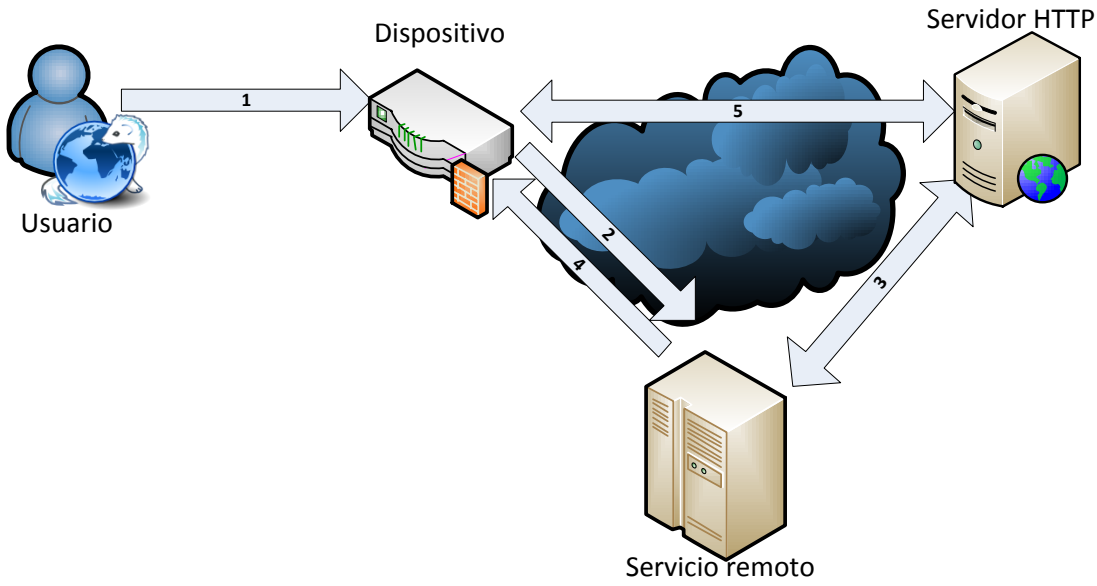
La solución que propone la empresa Open3s es una plataforma basada en la nube que consta de dos partes: un dispositivo físico que se le ofrece al usuario y se encarga de gestionar el tráfico y un servicio remoto encargado de toda la inteligencia.

La idea es que, cuando un usuario intente acceder a una web a través del dispositivo, éste consulte con un servicio remoto si debe permitir o denegar dicho acceso. Esto permitiría evitar las limitaciones de recursos propias de los dispositivos domésticos (potencia, memoria y almacenamiento) y permitiría poder analizar el contenido de la web para realizar, por ejemplo, un filtrado semántico, de las webs accedidas cosa totalmente fuera del alcance de computación de la mayoría de enrutadores domésticos.

El tener la inteligencia en la nube también nos permite tener que modificar una sola vez las configuraciones y que se automática y simultáneamente se apliquen a nivel global en los dispositivos (especialmente útil para colegios e institutos y familias con diversos domicilios), así como disponer de una copia de seguridad en caso que se deba remplazar el dispositivo o cambios en la plataforma por parte de Open3s sin necesidad de editar/acceder a los equipos de los clientes.

A continuación se muestra el funcionamiento de forma esquemática:

1. El usuario intenta acceder a una web.
2. El dispositivo comunica la petición y la referencia del usuario que la realizado a los servidores centrales (en adelante NUBE).
3. La nube se encargará de descargarse el mismo recurso y analizará para determinar si el perfil el usuario en cuestión tiene permiso para acceder al recurso solicitado.
4. Una vez finalizado el análisis el servicio contesta al dispositivo, en función del contenido del recurso y el rol del usuario, si debe permitir o no esa conexión.
5. Finalmente el dispositivo realiza la petición del recurso solicitado o bien responde al navegador con un mensaje de error indicándole al usuario que el recurso se encuentra filtrado.



Esquema de sincronización del dispositivo con la nube

En el caso que no se pueda acceder a la nube (por problemas técnicos de la propia nube o de la conexión entre el dispositivo y la nube), el dispositivo entraría en el modo "por defecto" y aplicaría a las conexiones una política por defecto

previamente configurada (aceptar todo, denegar todo o definir unas listas para aceptar o denegar).

3.3 Objetivos

Se deberá implementar un prototipo que implemente:

- El prototipo se implementará sobre una máquina virtual.
- Bastará con que el prototipo valide los usuarios localmente.
- El filtrado deberá estar basado en listas blancas y negras, se debe dejar de tal forma que sea fácil modificarlo en futuro para que implemente métodos de filtrado más complejos.
- El proxy debe ser capaz de interceptar tráfico HTTPS de la forma más transparente posible, sin que se moleste al usuario con alertas en el navegador y sin enmascarar los problemas de TLS/SSL con el servidor HTTPS.
- El proxy debe dejar un registro de las conexiones que se producen a través suyo y el resultado, para poderlas procesar más tarde.

El dispositivo tendrá que estar basado en software libre que permita su modificación y redistribución sin ningún tipo de problema.

El objetivo del proyecto es el de construir un prototipo del dispositivo descrito en el apartado anterior, para el proyecto nos centraremos en tener una maqueta que sea capaz de filtrar las peticiones en función del usuario, dado que el hardware puede ser la parte más cara y limitante del proyecto crearemos la maqueta en una máquina virtual y dejaremos la implementación en un dispositivo físico para cuando se disponga de un prototipo funcional.

Inicialmente se presentó otra propuesta de TFC, paralelamente a este, en el cual se implementaría un prototipo del servicio remoto y poder trabajar conjuntamente en un prototipo completo que incluyese la validación en remoto, pero dado que el proyecto de servicios en la nube se abandonó, se opta por una solución en la que el proxy valide los usuarios localmente.

Dado que tenemos que renviar la petición al servicio remoto, debemos de poder interceptar las comunicaciones cifradas mediante TLS/SSL del protocolo HTTPS, esto no es trivial, dado que el objetivo de TLS/SSL es precisamente ese, evitar que terceros intercepten comunicaciones entre el usuario y el servidor remoto.

También es importante para el servicio que se ofrece a los padres, dejar un registro de la actividad que tengan los usuarios para poder generar informes y estadísticas del uso que hacen sus hijos de Internet.

Obviamente, para poder tener control sobre la plataforma y poder realizar cambios, deberemos tener acceso al código y permiso para poderlo modificar y adaptar a nuestras necesidades, es decir que tendrá que estar licenciado bajo una licencia libre.

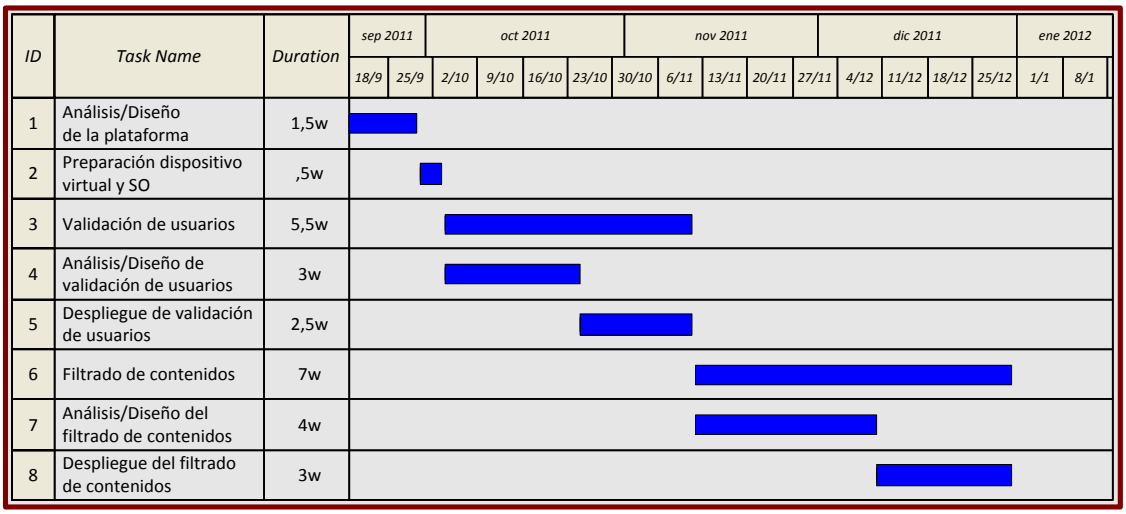
4 Planificación

Se pasa a planificar la dedicación en semanas que se le dedicarán al proyecto. Dicha planificación nos servirá principalmente para orientar sobre el estado del desarrollo del proyecto y si será posible de entregar en la fecha indicada con la dedicación prevista. La planificación se ha realizado teniendo en mente que se dedicarían cuatro horas al día, lo que nos daría unas veinte horas a la semana.

El proyecto se dividirñá en 4 tareas:

- **Análisis/diseño de la plataforma:** En esta tarea entraría todo lo que se ha explicado en el capítulo anterior, pensar en como se ha de comunicar el dispositivo con la plataforma y analizar sus pros y contras. Se planifica para esta tarea una semana y media.
- **Preparar el entorno:** Esta pequeña tarea consiste en montar la máquina virtual e instalar el sistema operativo así como la configuración de ese sistema operativo para adaptarlo lo máximo posible a un dispositivo físico de poca potencia. Se planifica para esta tarea media semana.
- **Diseñar e implementar la validación de usuarios:** Esta tarea representa todo el trabajo necesario para tener disponible un sistema de validación de usuarios que cumpla los objetivos anteriormente descritos. Se planifica para esta tarea cinco semanas y media. Podemos dividir esta tarea en dos subtareas: **análisis/diseño** (tres semanas) y **despliegue de la solución** (dos semanas y media).
- **Diseñar e implementar el filtrado de contenidos:** Esta tares es similar a la anterior y en ella incluiremos todo el trabajo necesario para tener

disponible un filtrado de contenidos que cumpla los objetivos descritos en el capítulo anterior. Se planifica para esta tarea siete semanas. También la dividiremos en dos subtareas: **análisis/diseño** (cuatro semanas) y **despliegue de la solución** (tres semanas).



5 Conceptos

5.1 TLS/SSL

5.1.1 Introducción

Protocolo TLS (Transport Layer Security), definido en el RFC 5246, y su predecesor el protocolo SSL (Secure Sockets Layer), definido en el RFC 6101, son protocolos criptográficos que pretenden proporcionar conexiones seguras a través de Internet usando protocolos de transporte fiables (por ejemplo TCP).

Estos protocolos proporcionan tres características a las comunicaciones a través de la red:

- **Confidencialidad:** O la prevención de la divulgación del contenido de los mensajes intercambiados a través de la red, de tal forma que sólo el emisor y el receptor conocerán el contenido del mensaje.
- **Integridad:** Evitar la manipulación de los datos durante el transporte a través de la red o, como mínimo, detectar esas manipulaciones para poder descartar dichos mensajes.
- **Autenticidad:** Tener la seguridad que el interlocutor del otro extremo es quien dice ser, es decir, evitar la suplantación de identidad.

5.1.2 Historia

SSL fue desarrollado por Netscape, se desarrollaron tres versiones aunque solo dos se publicaron (la versión 2.0 en 1995 y la versión 3.0 en 1996). La IETF publicó la especificación de SSL 3.0 en el RFC 6101.

TLS 1.0 fue publicado en 1999 como una actualización de SSL 3.0 en el RFC 2246 aunque ambos protocolos son incompatibles. En 2006 se publicó TLS 1.1 en el RFC 4346, en el que se corregían algunas debilidades de la especificación. TLS 1.2 se publicó en 2008 en el RFC 5246.

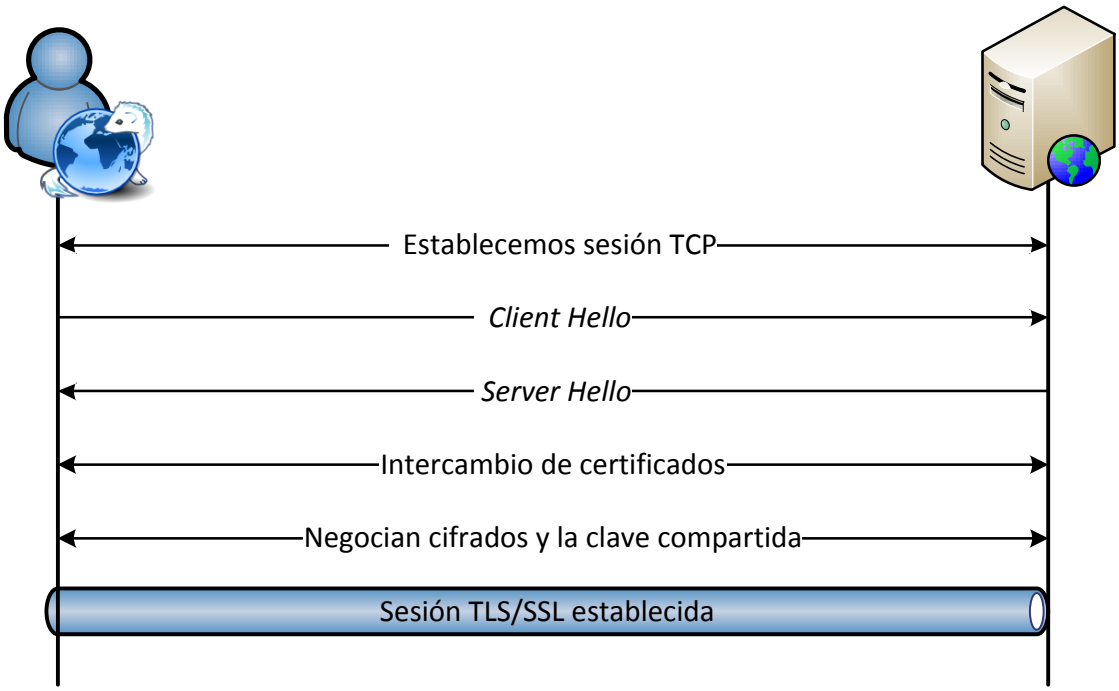
5.1.3 Funcionamiento

Primero de todo debemos establecer una conexión TLS/SSL, para ello seguimos el siguiente protocolo:

- Primero se establece una conexión confiable (por confiable nos referimos a un protocolo de transporte orientado a conexión, como por ejemplo TCP).
- Después el cliente inicia la conexión TLS enviando un *Client Hello* con la versión del protocolo más alta soportada y los algoritmos de cifrado y compresión soportados.
- El servidor le contesta con un *Server Hello* en el que el servidor escoge los parámetros entre los que le ha ofrecido el cliente en el *Client Hello*.
- Cuando acuerdan el tipo de cifrado y compresión a emplear intercambian certificados para confirmar que son quienes dicen ser, aunque en la mayoría de casos sólo envía certificado el servidor puede darse el caso en los que se requiere que el cliente también envíe el certificado.
- Una vez han validado la autenticidad de sus identidades, ambas partes pasan a negociar la clave empleada para cifrar y descifrar las comunicaciones en ambos extremos.

- Una vez ambos extremos se ponen de acuerdo en la versión de TLS/SSL, los cifrados y la clave que emplearán, el túnel está listo para transportar tráfico de forma segura.

De este modo se establece un canal que garantiza las tres características que describíamos al principio: confidencialidad (mediante los algoritmos de cifrado simétricos escogidos durante el handshake anteriormente explicado), integridad (mediante códigos de autenticación de mensajes) y autenticidad (mediante el uso de certificados X.509 para firmar los mensajes).



Handshake TLS/SSL

Hay que tener en cuenta que, pese a los intentos de crear un algoritmo seguro, a día de hoy todas las versiones de SSL y la versión 1.0 de TLS se consideran inseguras, debido a una vulnerabilidad del cifrado CBC. Pese a ello son todavía muchas las aplicaciones que todavía emplean versiones anteriores a TLS 1.1.

5.2 HTTP

5.2.1 Introducción

Protocolo de transferencia de hipertexto (HyperText Transfer Protocol) definido en los RFC: RFC 1945 (HTTP/1.0), RFC 2616 (HTTP/1.1), RFC 2774 (HTTP/1.2), es un protocolo de capa de aplicación empleado principalmente para la web (aunque dada su gran difusión, en algunos casos, se emplea para otro tipo de aplicaciones distribuidas).

5.2.2 Funcionamiento

El protocolo HTTP se basa en el modelo petición-respuesta, en cual el cliente inicia la comunicación enviando un mensaje (petición) al servidor, el servidor intenta realizar la acción solicitada por el cliente y le responde con el resultado de la acción (respuesta).

El protocolo HTTP utiliza los caracteres ASCII "retorno de carro" (CR, 0x0D) y "salto de línea" (LF, 0x0A) para indicar el final de línea. Y una línea en blanco (solamente los caracteres CR y LF) para indicar el fin del mensaje.

5.2.2.1 Petición

La forma de los mensajes de petición es la siguiente:

```
<método> <URI> HTTP/<versión><CR><LF>
<cabecera1>: <valor1><CR><LF>
<cabecera2>: <valor2><CR><LF>
(…)
<CR><LF>
```

Donde `<método>` es la acción que se pretende que realice el servidor, `<URI>` es el identificador del recurso sobre el que se pretende realizar la acción del método, el

campo `<versión>` se usa para asegurarnos que tanto el cliente como el servidor usan el mismo estándar, `<cabeceraN>` y `<valorN>` se emplean para pasarle más información al servidor sobre el contexto de la petición (que cliente realiza la petición, la fecha de la petición, en qué idioma preferimos la respuesta, si aceptamos que nos envíe la respuesta de forma comprimida, etc.).

Las acciones o peticiones que puede hacer el cliente (métodos) son, estos métodos pueden, o no, estar soportados por el servidor:

- **OPTIONS:** El cliente solicita al servidor una lista de métodos soportados sobre un URI concreto.
- **GET:** El cliente solicita una representación del URI solicitado. Éste es el método empleado cuando se solicita una página o una imagen en la web.
- **POST:** El cliente solicita que el servidor que procese los datos que se envían en el cuerpo de la petición. Éste método es el que se utiliza, por ejemplo, cuando envías información mediante un formulario en la web.
- **HEAD:** Este método solo nos retornará las cabeceras del URI sin el contenido. Éste método suele ser usado cuando se quiere comprobar la validez de un URI sin tener que transferir toda la información.
- **PUT:** Este método es empleado para subir el URI indicado al servidor.
- **DELETE:** Este método es empleado para eliminar el URI del servidor.
- **TRACE:** Este método nos retorna el camino seguido (proxys web) por la petición hasta llegar al servidor encargado de procesarlo.
- **CONNECT:** Se emplea para poder tunelar una conexión TCP mediante un proxy web.

5.2.2.2 Respuesta

La forma de los mensajes de respuesta es la siguiente:

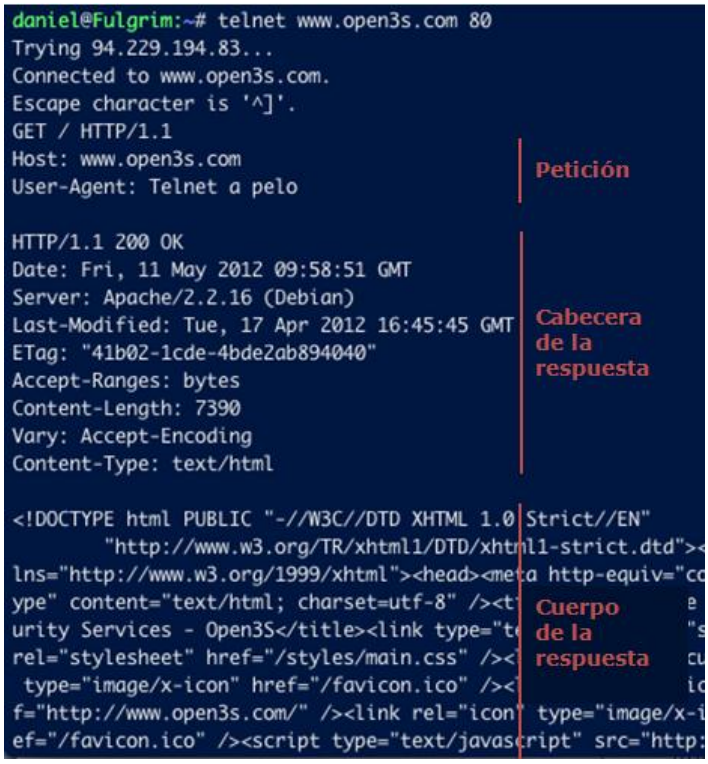
```
HTTP/<versión> <código> <descripción><CR><LF>
<cabecera1>: <valor1><CR><LF>
<cabecera2>: <valor2><CR><LF>
(…)
<CR><LF>
Respuesta del servidor.<CR><LF>
Normalmente suele ser HTML pero no ha de serlo
necesariamente.<CR><LF>
<CR><LF>
```

Donde de forma parecida a la petición, tenemos que `<versión>` nos indica qué conjunto de normas está siguiendo el servidor para enviarnos la información y que nosotros podamos entenderla correctamente, `<código>` nos indica el resultado del proceso solicitado (si ha finalizado correctamente, si ha fallado por culpa del cliente o del servidor, etc.), descripción es una representación en inglés de la información que nos aporta el código (para que sea más amigable con un usuario que accediera directamente al HTTP sin usar un cliente que le gestione la conexión), los pares `<cabeceraN>` y `<valorN>` nos dan información del contexto de la respuesta.

Los códigos de respuesta suelen ser tres dígitos decimales donde el dígito de mayor peso indica el tipo de respuesta y los dos dígitos de menor peso nos indican la respuesta en sí. Los rangos de respuesta son:

- **1xx:** Informativo, la petición se ha recibido y se continúa procesando.
- **2xx:** Éxito, la acción se ha recibido de forma correcta, se ha entendido y aceptado.
- **3xx:** Redirección, se deben realizar acciones adicionales por parte del cliente para completar la petición.

- **4xx:** Error del cliente, la petición contiene sintaxis incorrecta o no puede ser satisfecha.
- **5xx:** Error del servidor, el servidor falló al intentar satisfacer una petición aparentemente válida.



Ejemplo de petición HTTP

5.3 HTTPS

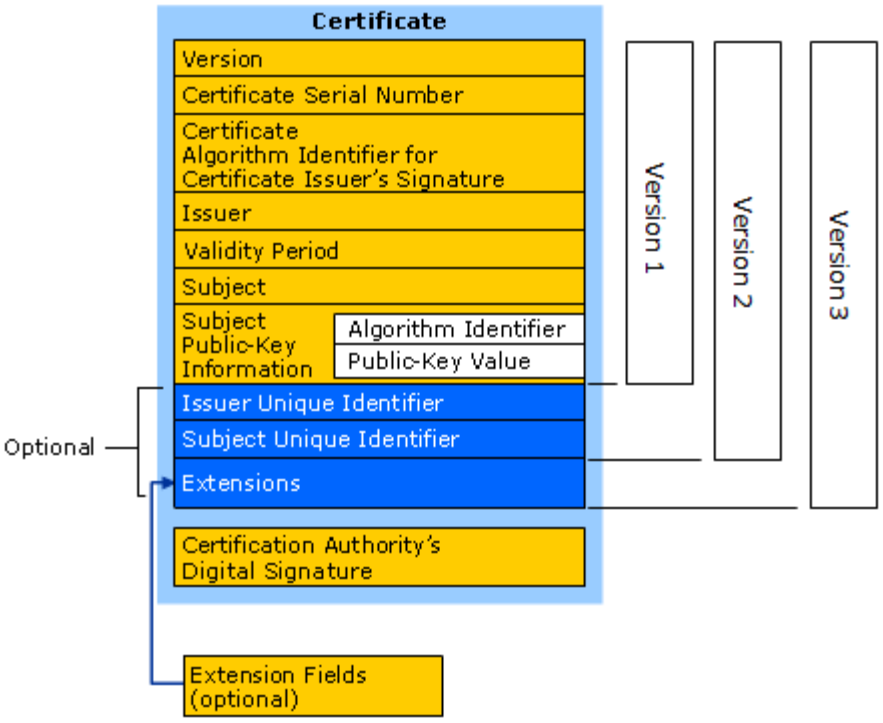
5.3.1 Introducción

Dado que la información que intercambia el protocolo HTTP viaja a través de texto plano, toda la información de las comunicaciones HTTP está disponible para el que pueda interceptarlas (alguien con acceso a algún dispositivo de red por donde pase la información, o cualquier usuario conectado a un Wi-Fi abierta, etc.). Para ello se definió en el 2000 se definió en el RFC 2818 el "HTTP sobre TLS" (o coloquialmente HTTPS, HyperText Transfer Protocol Secure), que básicamente consiste en establecer una sesión SSL/TLS sobre una conexión TCP antes de enviar

la información, de ese modo conseguimos que las comunicaciones entre el servidor y el cliente no han sido interceptadas ni modificadas durante su viaje.

5.3.2 Funcionamiento

El cliente se conecta con el servidor en un puerto TCP distinto que para HTTP (normalmente el 443), cada vez que se etablece una conexión TCP en ese puerto, se inicia la negociación TLS/SSL tal y como se explica en el apartado 2.1. A partir de este punto el cliente puede enviar la información a través de la conexión TLS/SSL del mismo modo que lo hace en HTTP, con la confianza que esa información no será interceptada ni manipulada en el viaje entre el cliente y el servidor (y viceversa).



Certificado X.509

El certificado X.509 que nos entrega el servidor contiene la clave pública con el que se firmarán las comunicaciones y debe de estar firmado a su vez por una entidad certificadora considerada de confianza por el navegador, y su Common Name (CN)

debe coincidir con el dominio al cual estamos realizando la petición. Esto nos garantizaría que realmente nos estamos conectando al servidor que queremos conectarnos.

5.3.3 Problemas de HTTPS

Dado que HTTPS es un protocolo ampliamente utilizado en Internet (y especialmente en las aplicaciones de Internet en las que se mueve dinero o información sensible) ha llamado la atención de hackers, investigadores de seguridad, cibercriminales y demás interesados en encontrar las debilidades de este protocolo.

Principalmente existen dos grandes problemas:

- La confianza real que ofrecen las entidades certificadoras (CA): Las CA que pueden firmar certificados válidos son seleccionadas por los desarrolladores del navegador. Éstos, a su vez, suelen ser bastante permisivos y aceptar un gran número de CA para evitar problemas a los usuarios. Esto nos lleva a que, por ejemplo, el gobierno chino pueda emitir un certificado válido en la mayoría de navegadores para www.facebook.com. Otro ejemplo podría ser que la seguridad de una CA válida se vea comprometida, los atacantes que tuviesen acceso a la clave privada de esa CA podrían emitir un certificado válido para cualquier dominio (hasta que el navegador no haya actualizado su lista de revocación de certificados (CRL)).
- La exactitud a la hora de escribir la URL: Como hemos comentado anteriormente, la validación del servidor al cual nos conectamos se realiza mediante el dominio de la URL. Esto ha dado lugar a una nueva modalidad

de ataque llamado phishing, el cual consiste en hacer creer al usuario que se está conectando al sitio que él quiere conectarse cuando en realidad se está conectando a un sitio fraudulento. Sería relativamente fácil registrar un dominio llamado www.pypal.com y solicitar a cualquier CA un certificado válido para ese dominio, una vez hemos creado un sitio a imagen y semejanza del sitio de PayPal en ese dominio sólo nos queda esperar que algún usuario despistado se equivoque al escribir la URL o solicitarle a algún usuario despistado que ingrese en esa URL. Desde el punto de vista del navegador, esa conexión sería "segura" dado que, efectivamente nos estamos conectando al servidor "legítimo" de www.pypal.com, dado que el navegador no puede saber que en realidad nuestra intención era conectar con www.paypal.com.

Source	Destination	Protocol	Info
10.211.55.3	69.171.224.37	TCP	49336 > 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
69.171.224.37	10.211.55.3	TCP	443 > 49336 [SYN, ACK] Seq=0 Ack=1 Win=32768 Len=0 MSS=1460 WS=2
10.211.55.3	69.171.224.37	TCP	49336 > 443 [ACK] Seq=1 Ack=1 Win=65700 Len=0
10.211.55.3	69.171.224.37	TLSv1	Client Hello
69.171.224.37	10.211.55.3	TCP	443 > 49336 [ACK] Seq=1 Ack=126 Win=32768 Len=0
69.171.224.37	10.211.55.3	TLSv1	Server Hello
69.171.224.37	10.211.55.3	TLSv1	Certificate, Server Hello Done
10.211.55.3	69.171.224.37	TCP	49336 > 443 [ACK] Seq=126 Ack=1993 Win=65700 Len=0
10.211.55.3	69.171.224.37	TLSv1	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
69.171.224.37	10.211.55.3	TCP	443 > 49336 [ACK] Seq=1993 Ack=312 Win=32768 Len=0
69.171.224.37	10.211.55.3	TLSv1	Change Cipher Spec, Encrypted Handshake Message
10.211.55.3	69.171.224.37	TCP	49336 > 443 [ACK] Seq=312 Ack=2040 Win=65652 Len=0
10.211.55.3	69.171.224.37	TLSv1	Application Data
69.171.224.37	10.211.55.3	TCP	443 > 49336 [ACK] Seq=2040 Ack=606 Win=32768 Len=0
69.171.224.37	10.211.55.3	TLSv1	Application Data

Captura de tráfico HTTPS con wireshark.

6 Validación de usuarios

6.1 Análisis

Dado que el filtrado de contenidos es dependiente del usuario que realiza la petición y que se debe guardar un registro del uso que hace cada usuario de Internet, es necesario identificar al usuario que realiza la petición así como limitar el acceso a los usuarios que no estén dados de alta en el sistema.

En un entorno en el que cada dispositivo fuese empleado por un solo usuario y que pudiésemos obtener un identificador único de cada dispositivo (por ejemplo, la dirección MAC o la dirección IP del dispositivo) este apartado sería mucho más sencillo. Pero lo más probable es que el dispositivo se emplee en redes en las que un mismo dispositivo sea compartido por diversos usuarios, por lo que tendremos que hacer que el usuario inicie sesión el usuario de forma explícita cuando quiera navegar a través del dispositivo.

6.1.1 Métodos de autenticación

Hay dos formas para que el usuario introduzca los credenciales para iniciar sesión mediante el método de HTTP *Basic Access Authentication* o mediante un portal cautivo. A continuación se desarrollan estos dos métodos.

6.1.1.1 Basic Access Authentication

En éste método, cuando el proxy recibe una petición de un cliente no autenticado en el sistema, responde con un código "`401 Authorization Required`" y añade una cabecera "`WWW-Authenticate: Basic realm="<MENSAJE>"`" a la respuesta. Cuando el navegador recibe esta respuesta del proxy, lanza un pop-up

con un formulario para que el usuario introduzca sus credenciales y una vez introducidas vuelve a enviar la misma petición añadiendo la cabecera "`Authorization: Basic <CREDENCIALES>`" con las credenciales que ha introducido el usuario codificadas en *base64*. Si esas credenciales son correctas, el dispositivo deja pasar la petición al servidor final y permitiendo la conexión normalmente, si no son correctas vuelve a responder "`401 Authorization Required`", este proceso se repite en todas las peticiones por lo que, normalmente, los navegadores recuerdan que direcciones de Internet le han pedido una *Basic Access Authentication* y realiza las peticiones para esa dirección con la cabecera "`Authorization: Basic <CREDENCIALES>`" automáticamente evitando tener que lanzar un pop-up de validación a cada petición.

Este tipo de validación es bastante sencillo, pero tiene dos principales inconvenientes:

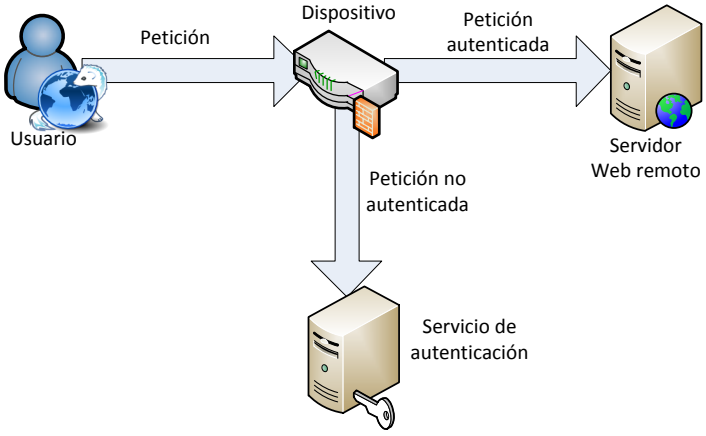
- El pop-up que lanza el navegador cuando recibe este tipo de respuestas no suele ser demasiado explicativo, lo que puede llevar a desorientar al usuario.
- Si el servidor final también emplea *Basic Access Authentication* puede que la autenticación del dispositivo enmascare la autenticación del servidor final, impidiendo su funcionamiento.

6.1.1.2 Portal cautivo

El portal cautivo es un método de autenticación de usuarios de red que funciona de la siguiente manera: El usuario intenta acceder a un recurso de la red, el dispositivo intercepta esa conexión y la redirige a un portal con un formulario que

el que se invita al usuario a ingresar sus credenciales, aunque también se puede aprovechar para poner información que le permita al usuario comprender qué está pasando y por qué tiene que introducir las credenciales, además también se puede añadir un enlace u otro métodos de contacto con el servicio técnico. Este portal será lo único a lo que pueda acceder el usuario hasta que no se valide correctamente. Una vez el usuario se valide correctamente en el portal, el sistema debe recordar a ese usuario y no volver a molestarle con el portal cautivo (al menos durante un tiempo razonable de tiempo).

Este método es el más empleado en servicios de acceso a redes wireless públicas o privadas, dado que muchas soluciones de portal cautivo también implementan un registro del tráfico cursado por los usuarios.



Esquema de funcionamiento del portal cautivo

6.1.1.3 Método empleado

Vistos los problemas de incompatibilidad que tiene el *Basic Access Authentication* con los servidores que empleen el mismo método, además de la poca información que aportan los navegadores al usuario cuando reciben una respuesta "`401 Authorization Required`" que puede desconcertar al usuario, queda claro que tendremos que emplear un portal cautivo.

6.1.2 Soluciones existentes

Seguidamente analizaremos las soluciones de código abierto de portal cautivo que existen en el mercado.

6.1.2.1 CoovaChili

Portal cautivo basado en el código del descontinuado ChilliSpot, está mantenido por la empresa CoovaTechnologies. Es un producto bastante completo que cuenta con sistema de autenticación y contabilidad bastante potente basado en RADIUS. También es interesante la cantidad de plugins y desarrollos de terceros para integrarlo con otros productos (Drupal, Openmoko, Cisco Aironet, Ubiquiti, etc.).

6.1.2.2 WiFiDog

WiFiDog es un portal cautivo de código abierto desarrollado por la comunidad de Île Sans Fil. WiFiDog consta de dos partes, la puerta de enlace y el servidor de autenticación. La puerta de enlace es la programa encargado de permitir o denegar el tráfico HTTP en función de si el usuario está autenticado o no, dicho programa está escrito en C y ocupa menos de 15kb, lo cual lo hace ideal para funcionar en dispositivos empotrados. El servidor de autenticación está escrito en PHP y necesita una base de datos PostgreSQL para funcionar, por lo que se debería desplegar un servidor a parte del dispositivo.

6.1.2.3 NoCatAuth

NoCatAuth es un portal cautivo escrito en Perl relativamente sencillo, sin necesidad de depender de servidores externos, el proyecto lleva parado desde 2004. Básicamente consiste en un servidor ligero con un CGI que se encarga de gestionar reglas de *iptables*.

6.2 Diseño

6.2.1 Problemas en al desplegar solucione de portal cautivo

En un primer momento se pensó en CoovaChilli como el sistema de portal cautivo a desplegar en el dispositivo, dado que es un software bastante completo que incluye una parte de registro de actividad de la red por cada usuario, así como bastantes plugins y desarrollos de terceros para integrarlos con otros productos.

CoovaChilli emplea el protocolo RADIUS para validar los usuarios, esto implica la necesidad de un servidor de RADIUS contra el cual realizaría las consultas el portal cautivo. En principio se pensó en instalar el servidor de RADIUS en el dispositivo y que fuese sincronizando periódicamente los usuarios con la nube.

Cuando se desplegó el servidor de RADIUS FreeRadius, se vio claramente que está pensado para ser desplegado en un servidor centralizado contra el cual los demás dispositivos realizan las consultas y no para ser desplegado en los dispositivos directamente, por el alto consumo de memoria que nos consume.

La opción de desplegar el servidor de RADIUS en la nube y que el dispositivo valide contra la nube en cada acceso no es aceptable, dado que queremos que el usuario se pueda validar aunque falle la conectividad con la nube.

La siguiente opción fue WiFiDog, aunque no sea tan empleada a nivel comercial también está bastante extendido. También se realizó una maqueta con WiFiDog, y la verdad que el módulo gateway (que es el encargado de redirigir las peticiones) sorprendió gratamente, dado que apenas ocupaba espacio. El problema se volvimó

a tener con la parte de validación de usuarios, de forma parecida a como lo hacía CoovaChilli, aunque en vez de emplear RADIUS emplea un protocolo propio que depende de una base de datos PostgeSQL, esto nos traería problemas a la hora de sincronizar la lista de usuarios del dispositivo con la nube.

6.2.2 Nuestro propio portal cautivo

Dado que no se encontraron soluciones de portal cautivo que se adaptaran a las necesidades del proyecto y que simplemente es necesario que implemente la validación de usuarios (del registro de uso de la red ya se encarga el proxy), se decidió que lo mejor sería implementar nuestro propio portal cautivo.

El portal cautivo se montará con reglas de Netfilter que serán gestionadas por un CGI sobre un servidor ligero como *Lighttpd* o *nginx*. La idea es que una regla genérica cambie la dirección IP destino de todas las conexiones HTTP y HTTPS (es decir, los paquetes TCP con puerto de destino 80 y 443 respectivamente) que entren por alguna de las interfaces de red a las que se conectan los usuarios, a la dirección de localhost (127.0.0.1). En esa dirección estará escuchando el servidor HTTP que hospeda el portal cautivo y que le ofrecerá al usuario la posibilidad de iniciar sesión. Una vez el usuario haya logrado iniciar sesión, el portal añadirá una regla a Netfilter específica para la dirección IP que originó la conexión.

Finalmente se tendrá que implementar algún proceso que vaya comprobando las reglas que no se estén usando para darlas de baja, es decir, caducar la sesión de ese usuario.

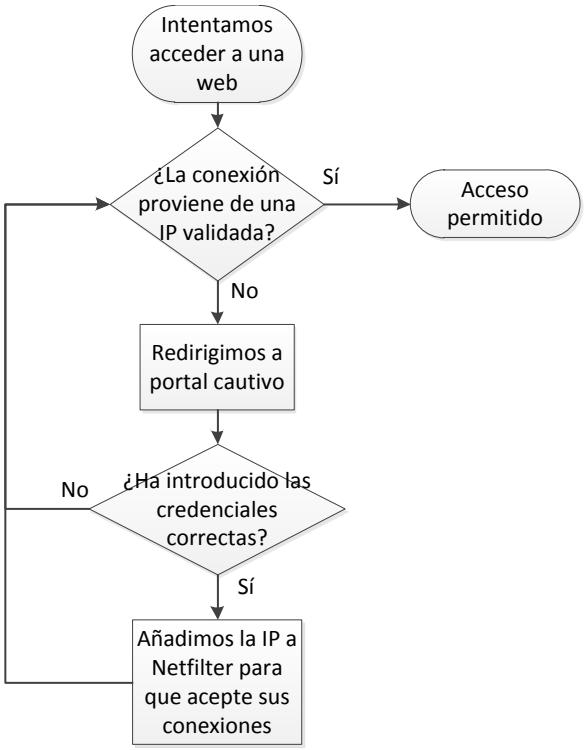
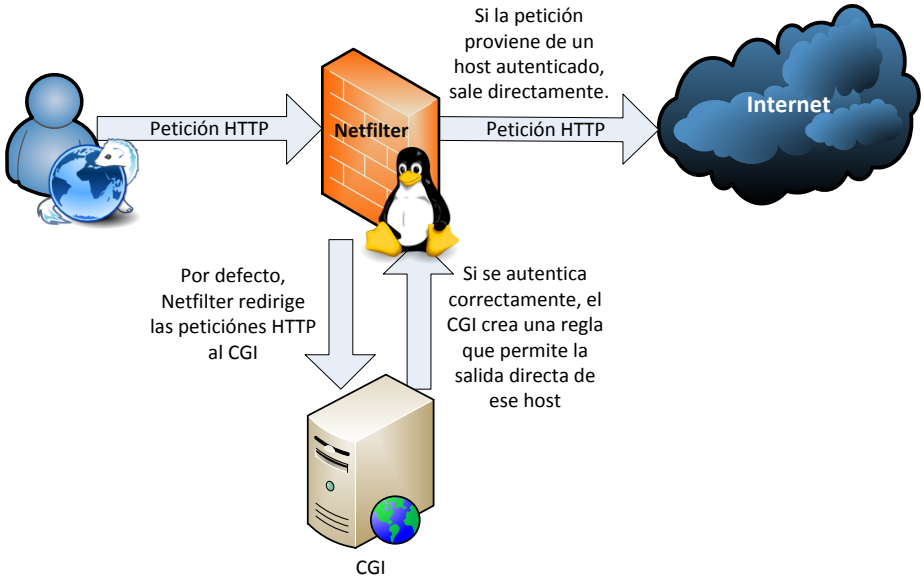


Diagrama de funcionamiento de un portal cautivo

6.3 Implementación

Como se explica en el apartado anterior, dado que no se encontraron soluciones de portal cautivo de código abierto y que se adecuasen a las necesidades del proyecto, se ha decidido implementar nuestro propio portal cautivo.



Ejemplo de funcionamiento de Chell

Se puede dividir el portal cautivo en cuatro funcionalidades a resolver: la redirección de paquetes, el "inicio de sesión" por parte del usuario y la caducidad de las "sesiones".

6.3.1 Redirección de paquetes

Para redirigir las peticiones emplearemos Netfilter que es el módulo de Linux que nos permite interceptar y manipular paquetes de red. Para crear y editar las reglas de Netfilter se emplea iptables, que es la herramienta empleada modificar las reglas que nos viene integrada en la mayoría de distribuciones (o, al menos, en los repositorios).

La idea es que se cree una regla genérica que redirija todos los paquetes con puerto de destino 80 (HTTP) o 443 (HTTPS) hasta el servidor HTTP que albergue el CGI:

```
# /sbin/iptables –t nat -I PREROUTING\
   -i ${LAN_IFACE} -p tcp -m multiport --dports 80,443\
   -j DNAT --to-destination 127.0.0.1
```

Donde `${LAN_IFACE}` es la interfaz de red donde se conectan los usuarios.

Cuando los usuarios se validen correctamente en el sistema, se deberá crear una regla con más prioridad que la anterior que permita el tráfico de forma normal para la dirección IP de origen desde donde se validó correctamente en el sistema.

```
# /sbin/iptables –t nat -I PREROUTING\
   -s ${IP_ORIGEN} -i ${LAN_IFACE} -p tcp -m tcp --dport 80\
   -j REDIRECT –to-port ${PUERTO_PROXY_HTTP}
# /sbin/iptables –t nat -I PREROUTING\
   -s ${IP_ORIGEN} -i ${LAN_IFACE} -p tcp -m tcp --dport 443\
   -j REDIRECT –to-port ${PUERTO_PROXY_HTTPS}
```

Donde `${PUERTO_PROXY_HTTP}` y `${PUERTO_PROXY_HTTPS}` son los puertos donde estará escuchando el proxy de filtrado para las conexiones HTTP (TCP80) y HTTPS (TCP443) respectivamente.

6.3.2 Validando al usuario o "Inicio de sesión" .

Mientras que el usario no se valide correctamente en el sistema, sus peticiones HTTP y HTTPS serán redirigidas a un servidor HTTP ligero (Lighttpd) que tendremos escuchando en la misma máquina. En dicho servidor se estará ejecutando un CGI que hará de portal cautivo.

Lo primero que se debe hacer, para evitarle confusiones al usuario, es configurar dos virtualhosts: el por defecto y el del portal cautivo. El virtualhost por defecto estará configurado para que ante cualquier petición que no esté dirgida al dominio del portal cautivo, responda con una redirección al dominio del portal cautivo. En el virtualhost del portal cautivo se encontrará el portal en el cual el usuario se podrá validar en el sistema. Este modo se evitará que el usuario confunda el formulario del portal cautivo con la web a la que quería acceder.

Una vez configurado el servidor para que gestione los dos virtualhosts se deberá pasar a programar el CGI encargado de la validación. Para el proyecto se almacenarán los usuarios en memoria, que serán cargados de un fichero en texto plano, aunque es deseable para el futuro emplear algún sistema de base de datos ligero, como SQLite.

Para evitar que alguien con acceso al dispositivo pueda obtener las contraseñas de los usuarios, se deberá evitar almacenar las contraseñas en el fichero. En vez de

eso se almacerán los “hashes” lo que haría que no se pudiesen obtener las contraseñas

6.3.3 Caducidad de las sesiones.

Hasta este punto se tiene un sistema de portal cautivo en el que los usuarios han de validarse (o iniciar sesión), a no ser que accedan desde una dirección IP que haya sido dada de alta con aterioridad (ya sea porque está empleando el mismo equipo que ya empleó otro usuario validado con anterioridad, o porque el DHCP le asignó a su equipo una dirección IP ya validada anteriormente) lo que permitiría que el usuario pasara por el proxy sin necesidad de validarse.

Para evitar esto se deberán eliminar las reglas que no se hayan usado durante un períodod tiempo. Esto se puede conseguir con un script en el cron que se ejecute cada, por ejemplo, 15 minutos y que se encargue de eliminar las reglas que no han tenido actividad desde la última vez que se ejecutó.

```
#!/bin/bash

export PATH="/bin:/sbin/usr/bin:/usr/sbin"

# Seleccionamos las reglas de la tabla de nat que no han tenido
#actividad en el último período (excluyendo la regla general) y
#las eliminamos.
iptables-save -t nat -c | grep '^\[' | head -n -2 |\
              grep '^\[0:0\]' | grep -o '\PREROUTING.*' |\
              xargs -I{} iptables -t nat -D {}

# Ponemos los contadores a 0
iptables -t nat -Z PREROUTING
```

7 Filtrado de comunicaciones HTTP

7.1 Análisis

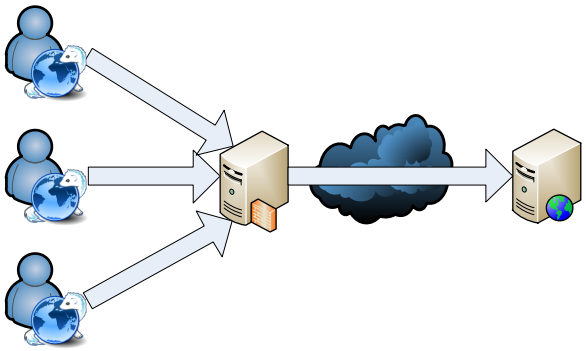
Aquí se ha de encontrar una solución que permita interceptar las comunicaciones HTTP/HTTPS, para que puedan ser analizadas y decidir si se debe permitir o no dicha comunicación. Se puede desglosar el filtrado de comunicaciones HTTP/HTTPS en tres partes: proxy HTTP, filtrado de contenidos y HTTP sobre TLS/SSL.

7.1.1 Proxy HTTP

Para realizar la tarea de filtrado emplearemos un servidor proxy, es decir un servidor que actúa en representación de otro. En el caso de un servidor proxy HTTP lo que hace es escuchar peticiones HTTP, renviar esa petición a un servidor remoto y, cuando el servidor remoto contesta, devolverle la respuesta al navegador que inició la petición.

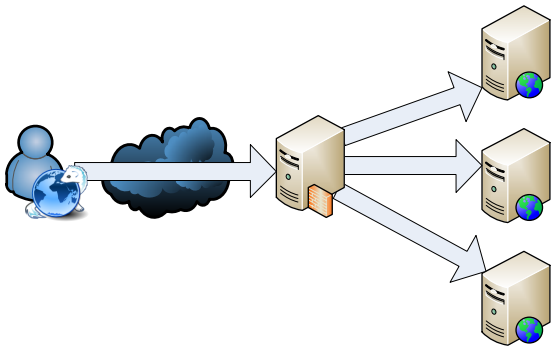
En función de su posición respecto el servidor web y el usuario se pueden distinguir dos tipos de proxy: directo e inverso.

- **Directo:** En esta configuración el proxy se sitúa delante de los usuarios, por lo general se emplea para controlar el uso de Internet de los usuarios de una organización así como optimizar el ancho de banda, dado que la mayoría de estos proxies implementan una caché que almacena un recurso durante un tiempo, así si otro usuario accede a una URL a la que ya habían accedido recientemente, el proxy se limitará a servirle el recurso de la caché evitando una petición a través de Internet.



Proxy HTTP directo

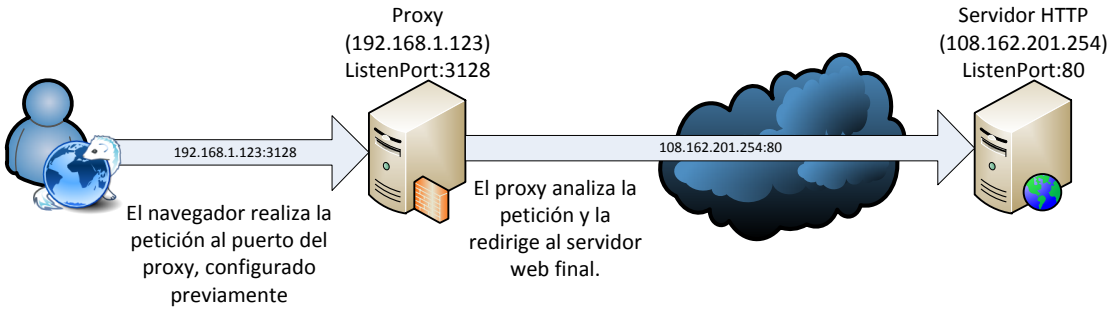
- **Inverso:** En esta configuración el proxy se coloca delante del servidor (o servidores) HTTP. El objetivo suele ser el de optimizar la carga de los servidores HTTP mediante la caché (sobretodo de contenido dinámico que no varía demasiado a menudo, pero que genera una carga en el sistema) y también para el balanceo de carga entre varios servidores HTTP



Proxy HTTP inverso

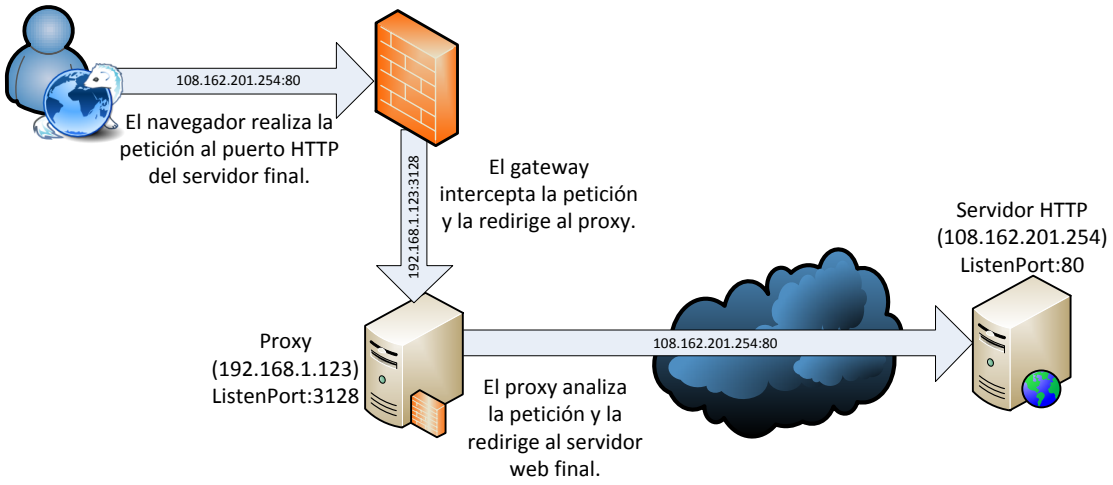
Obviamente, el proxy que se debrá implementar en este proyecto debe ser de tipo directo. Y dentro de los proxies directos, se pueden distinguir dos tipos dependiendo de si el navegador es consciente o no de que el proxy está "interceptando" la comunicación, podemos hablar de dos tipos de proxy: explicitos y transparente.

- **modo explícito**: En el cual se debe configurar el navegador para que acceda al proxy.



Proxy HTTP explícito

- **modo transparente:** En el cual no hace falta configurar nada, dado que en algún punto de la ruta hasta el servidor final, un firewall redirige la petición hasta el servidor proxy.

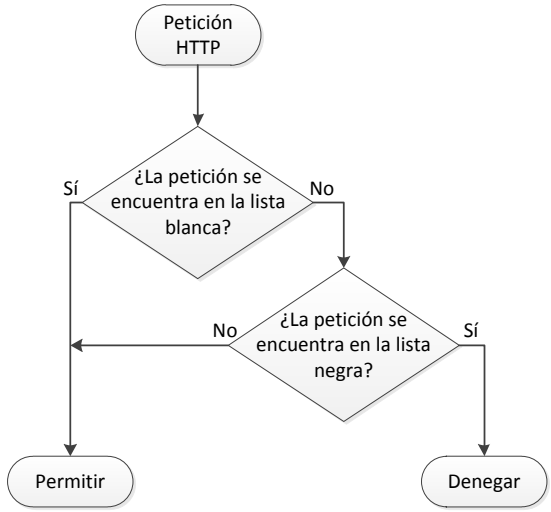


Proxy HTTP transparente

En este caaso, el proxy debe funcionar en modo transparente, dado que queremos simplificar al máximo la instalación al usuario final. Para ello se deberan emplear reglas de firewall (iptables/Netfilter en el caso de Linux) para redirigir todas las conexiones HTTP hacia el proxy, tal y como se vió en el apartado anterior.

7.1.2 Filtrado mediante listas

Como se han definido en los objetivos del proyecto, de momento bastará con que el proxy implemente el filtrado con listas. Este tipo de filtrado consta de diversas listas: normalmente una llamada blanca y otra llamada negra.



Esquema de funcionamiento de las listas blancas y negras

- Si el dominio o URL de la petición HTTP se encuentra en la lista blanca, la petición se permite.
- Si el dominio o URL de la petición HTTP no se encuentra en la lista blanca y se encuentra en la lista negra, la petición es denegada.
- Si el dominio o URL de la petición HTTP no se encuentra en la lista blanca ni en la lista negra se aplica la acción por defecto, que normalmente suele ser aceptar la petición.

Dicho método sólo protegería de las webs que se hayan incluido en la lista negra, por lo que seguro que se dejarían accesibles miles de webs "perniciosas", una forma de evitar esto sería el modificar el algoritmo para que la acción por defecto sea la de denegar el acceso, por lo que sólo estarían disponibles las webs incluidas en la lista blanca, lo que daría una experiencia bastante pobre de Internet. Es por

ello que se debe implementar una solución en la que sea fácil de modificar en un futuro para que puedan emplear otros métodos más complejos y efectivos para el filtrado de peticiones HTTP.

7.1.3 HTTP sobre TLS/SSL

El tema de proxies HTTP con filtrado es bastante sencillo y está bien resuelto, una vez te llega la petición, la examinas y decides si la dejas pasar o no. Al ser texto plano no hay ningún problema.

Los problemas aparecen cuando se introduce TLS/SSL en las comunicaciones, dado que, precisamente, la finalidad de estos protocolos es la de evitar que terceros puedan fisgar el contenido de las comunicaciones. En la mayor parte de proxies HTTP esto se soluciona con tunelando la petición HTTPS en una petición HTTP, en la cual el navegador realiza una petición al proxy con el método *CONNECT*, como cabecera la dirección a la cual se quiere conectar y como datos enviaría la petición HTTPS completa. En este caso el proxy examinaría las cabeceras de la petición *CONNECT* y renviaría la petición HTTPS al servidor de destino, pero sin ser consciente de qué hay dentro de ella. Obviamente para que éste método funcione correctamente el navegador ha de ser consciente que la petición debe atravesar un proxy y poder tunelar la petición mediante *CONNECT* por lo que no sirve para este proyecto.

7.1.4 Soluciones existentes

7.1.4.1 Squid Proxy Caché

Squid es el proxy HTTP de código abierto por excelencia, capaz de funcionar en modo directo como en modo inverso. Probablemente sea el proxy más empleado en entornos empresariales.

Una de las mayores ventajas para el proyecto es la posibilidad de configurar el proxy para que inspeccione las peticiones HTTPS, lo que ellos llaman "Squid In The Middle", que consiste en que Squid es capaz de crear un certificado SSL dinámicamente con una CA previamente definida (y que se debería de configurar en el navegador cliente como CA autorizada), de este modo el navegador considerará que la conexión es legítima.

Desgraciadamente éste método sólo funciona si se configura el proxy en modo explícito, y aunque tienen en la lista de tareas pendientes el poder configurar "Squid In The Middle" en modo transparente, es poco probable que puedan tenerlo acabado para la fecha de entrega del proyecto.

Otra parte negativa de Squid es el hecho que no es un proxy diseñado para dispositivos empotrados precisamente, dado que consume grandes cantidades de memoria y hace el uso intensivo de disco (lo que reduciría drásticamente la vida de la memoria flash que emplean los dispositivos empotrados).

7.1.4.2 TinyProxy

TinyProxy es un proxy diseñado para ser ligero (lo que es interesante para nuestro proyecto). Soporta HTTP y HTTPS (empleando el método *CONNECT*), proxy en

modo transparente (aunque no para HTTPS, dado que emplea el método *CONNECT* para ello).

Otra de las ventajas de este proxy es la simplicidad de su código, lo que lo que facilitaría la modificación de su código para incluir métodos de filtrado más elaborados en un futuro.

Desgraciadamente este proxy tampoco soporta el filtrado HTTPS en modo transparente, ni, obviamente, inspeccionar el tráfico SSL.

7.2 Diseño

7.2.1 Primeros intentos

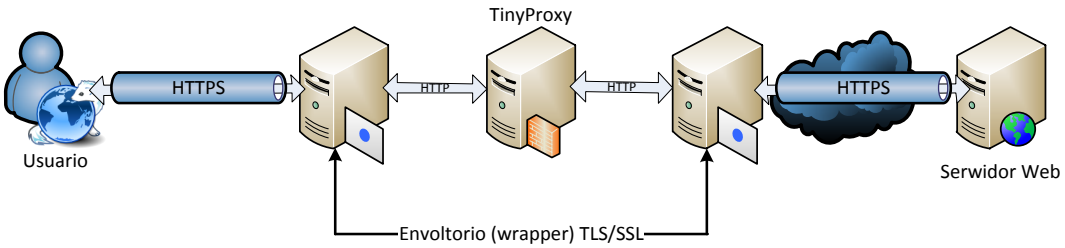
Cuando se planteó este proyecto lo primero que se pensó fue en emplear Squid para el tema del filtrado mediante listas, dado que era una solución conocida y con la que se había trabajado anteriormente por lo que hacía que la curva de aprendizaje ya estuviese superada. No obstante se vió que el uso que hacía del disco para la caché era muy agresivo, y eso acortaría la vida útil de las memorias flash que tienen los dispositivos, por lo que se deshechó Squid bastante pronto.

El siguiente candidato fue tinyproxy, un proxy HTTP escrito en C con la idea de der pequeño en disco y ligero en memoria, lo que lo hace ideal para un dispositivo limitado como el que se tiene en mente para el proyecto. Además al leer el código se nota que fue diseñado con la simplicidad en mente, por lo que resultaría relativamente fácil añadir modificaciones al código cuabdo fuera necesario.

Por lo que se desplegó en un primer momento una instancia de TinyProxy para el apartado de filtrado de peticiones. Tras unos pocos minutos de configuración ya se tenía funcionando un proxy HTTP/HTTPS con filtrado por listas. Y con un par de reglas de Netfilter se tenía listo el proxy HTTP en modo transparente, pero no había soporte para HTTPS en modo transparente.

7.2.2 Envoltorios para TLS/SSL

Con TinyProxy desplegado correctamente con HTTP en modo transparente, se intentó crear un par de programas encargados de establecer las sesiones TLS/SSL con el navegador y el servidor web final. De este modo, tanto para el navegador como para el servidor web se estaría estableciendo una sesión TLS/SSL, pero para el proxy se estaría enviando todo en HTTP plano.



Durante un tiempo el el filtrado de contenidos se empezó a implementar siguiendo este diseño, hasta que se vió que era bastante complejo y demasadio "sucio" (dado que se trataba más de un "hack" o improvisación que un producto de ingeniería).

7.2.3 Nuestro propio proxy HTTP/HTTPS

Como se ha comentado durante un tiempo se tuvo la idea de desplegar una instancia de TinyProxy con un par de programas implementados en Python que le tradujesen de HTTPS a HTTP, y luego volviesen a traducidr el HTTP a HTTPS.

Pero esta aproximación era bastante poco (o nada) elegante, le estábamos añadiendo una serie de componentes que parecían parches. Así que viendo que en este apartado tambiémn se tendrían que implementar nuevos componentes, se decidió de implemtnar el proxy entero, con soporte para HTTPS en modo transparente de forma nativa.

7.3 Implementación

Dado que no hemos encontrado ninguna solución de proxy HTTP que implemente "Man In The Middle" en modo transparente, deberemos implementar nuestro propio proxy. Como esto no entraba dentro de los objetivos del proyecto, y puede alargarlo bastante, el prototipo se escribirá en Python, que es un lenguaje de scripting potente y con una gran colección de bibliotecas que le permite desarrollar una gran cantidad de cosas de serie. El proxy deberá de poder realizar:

7.3.1 Proxy de peticiones HTTP en modo transparente

Dado el perfil no técnico de los usuarios del producto, es importante que se no tenga que configurar manualmente cada dispositivo para poder funcionar con el dispositivo o, al menos, minimizar la configuración necesaria. En este caso hacer que el usuario tenga que configurar cada aplicación que desee comunicarse por HTTP/HTTPS es excesivo, así como a veces imposible, dado que existen aplicaciones en las que no se les puede indicar que empleen un proxy.

Todo esto quiere decir que el dispositivo debe implementar una regla de firewall que renvíe todas las peticiones HTTP/HTTPS al puerto local donde esté escuchando el proxy. Para ello que empleamos iptables y deberemos emplear los siguientes comandos:

```
# iptables -t nat -I PREROUTING\
    -i ${LAN_IFACE} -p tcp -m tcp --dport 80\
    -j REDIRECT --to-port ${PROXY_HTTP}

# iptables -t nat -I PREROUTING\
    -i ${LAN_IFACE} -p tcp -m tcp --dport 443\
    -j REDIRECT --to-port ${PROXY_HTTPS}
```

Donde `${LAN_IFACE}` es el nombre de la interfaz donde se conectan los usuarios. Y `${PROXY_HTTP}` y `${PROXY_HTTPS}` son los puertos donde escucha el proxy para HTTP y HTTPS. Estas reglas lo que harán será renviar todo paquete que nos entre por la interfaz "LAN" cuyo puerto de destino sea el 80 (HTTP) o 443 (HTTPS) a los puertos donde esté el proxy escuchando.

Aunque dado que estas reglas las tiene que crear el portal cautivo que hemos visto anteriormente y sólo deben aplicar al cliente que se haya validado correctamente, por lo que deberemos crearlas algo más específicas:

```
# iptables -t nat -I PREROUTING\
   -s ${VALIDATED_IP} -i ${LAN_IFACE} -p tcp -m tcp --dport 80\
   -j REDIRECT --to-port ${PROXY_HTTP}

# iptables -t nat -I PREROUTING\
   -s ${VALIDATED_IP} -i ${LAN_IFACE} -p tcp -m tcp --dport 443\
   -j REDIRECT --to-port ${PROXY_HTTPS}
```

Donde `${VALIDATED_IP}` sea la dirección IP del host que se haya validado correctamente en el portal cautivo. También es importante que estas reglas se añadan al principio de la lista (con el flag `-I`)

Una vez el paquete llega al proxy tenemos que poder averiguar a qué dirección IP iba destinada la conexión en un primer momento. La mayoría de los proxies obtienen esta información examinando la cabecera "*Host*" de la petición HTTP, no nos sirve dado que cuando queramos establecer sesiones HTTPS (ver más

adelante) necesitaremos saber la dirección original para establecer los túneles TLS/SSL antes de gestionar el tráfico HTTP.

En Linux podemos obtener la dirección IP del paquete antes de hacer el *REDIRECT* con la opción `SO_ORIGINAL_DST` de la función `getsocketopt`.

```
import struct

def get_original_address (sock):
    '''Obtiene la dirección destino original, antes de aplicar
    el filtro NAT con netfilter.

    @param sock objeto socket de la conexión entrante.
    @return Tupla con la dirección IP y el puerto al que iba
            dirigida la conexión originalmente.'''

    SO_ORIGINAL_DST = 80
    dst = sock.getsockopt(socket.SOL_IP, SO_ORIGINAL_DST, 16)
    _, dst_port, oct1, oct2, oct3, oct4 = struct.unpack(
                                                "!HHBBBB8x", dst)
    ip = "%s.%s.%s.%s" % (oct1, oct2, oct3, oct4)
    return (ip, dst_port)
```

Una vez tenemos la dirección del servidor final, podemos conectarnos a él y simplemente renviar todo lo que recibamos el navegador. Aunque, debido a que el proxy debe generar un log donde consten las conexiones que realiza cada usuario, deberemos "entender" las peticiones y respuestas que renviemos para poder generar un registro de esas comunicaciones. Afortunadamente Python viene con el módulo BaseHTTPServer que nos permitirá crear la interfaz entre el navegador y el proxy con poco código. Para crear un servidor web que escuche las peticiones HTTP basta con:

```
from SocketServer import ThreadingMixIn
from BaseHTTPServer import HTTPServer
from BaseHTTPServer import BaseHTTPRequestHandler

class ProxyHTTPHandler (BaseHTTPServer.BaseHTTPRequestHandler):
    '''Aquí gestionaríamos las peticiones, nos bastará con
```

```
    implementar el método do_${HTTP_METHOD} para cada acción
    HTTP que queramos soportar. En dicho método tendríamos que
    implementar la lógica necesaria para renviar la petición'''
    pass

class Threaded_HTTP_Server (ThreadingMixIn, HTTPServer):
    '''Al heredar de la clase ThreadingMixIn, hacemos que
    HTTPServer gestione las peticiones en hilos'''
    daemon_threads = True
    allow_reuse_address = True

http_server = Threaded_HTTP_Server((bind_addr, http_port),
                                   ProxyHTTPHandler)
```

Con éste código creamos un objeto `http_server` el cual se encargará de lanzar un nuevo hilo y pasarle la petición a la clase `ProxyHTTPHandler` que se encargará de gestionar dicha petición, en nuestro caso, renviar la petición al servidor final.

La parte "interesante" del proxy estará en la clase `ProxyHTTPHandler`, que se encargará de entender la petición HTTP (tarea que realiza la clase BaseHTTPRequestHandler), recolectar la información necesaria para el registro y renviarla al servidor final.

```
from socket import socket
from socket import AF_INET, SOCK_STREAM
from BaseHTTPServer import BaseHTTPRequestHandler

class ProxyHTTPHandler (BaseHTTPRequestHandler):
    CRLF = '\r\n'

    def __init__ (self, request, client_address, server):
        '''Esta clase se instancia cuando http_server recibe una
        conexión, dado que la conexión está "nateada" deberemos
        obtener la dirección IP destino original del paquete,
        una vez tenemos esa dirección estableceremos una
        conexión hacia ese servidor para poder renviarle las
        peticiones.

        @param self El objeto del que forma parte el método.
        @param request Socket que repesenta la conexión entre el
                       cliente y el proxy
        @param client_address Dirección IP y puerto del cliente.
        @param server Objeto HTTPServer que nos ha pasado la
                      petición.'''
```

```
        self.request = request
        self.protocol = 'http'
        #Obtenemos la dirección IP destiono original.
        addr = self.get_original_address()
        #Establecemos una conexión con el servidor final.
        self.target = socket(AF_INET, SOCK_STREAM)
        self.target.connect(addr)
        self.target.settimeout(2)
        #Definimos los filtros que se aplicarán a esta petición.
        self.filters = [ WhiteListFilter(), BlackListFilter() ]
        #Llamamos a la clase madre para que haga la "magia"
        BaseHTTPRequestHandler.__init__(self, request,
                                        client_address, server)
    def do_request (self):
        '''Método que gestiona el renvio de peticiones'''
        # La clase BaseHTTPRequestHandler nos proporciona la
        #información de la petición HTTP ya prepada y
        #"desmenuzada" :)
        command = "%s %s %s" % (self.command, self.path,
                                self.request_version)
        command += self.CRLF
        self.target.sendall(command)
        allowed = "Exception"
        #Obtenemos el nombre del host para poderlo registrar.
        if 'host' in self.headers:
            host = self.headers['host']
        else:
            host = 'Nohost'
        try:
            # Aplicamos los filtros
            if self.allowed():
                allowed = "Allowed"
                #Enviamos la cabecera de la petición.
                self.send_headers()
                if self.command == 'POST':
                    #En caso que la petición sea un POST,
                    #renviamos el POSTDATA
                    self.forward_postdata()
                #Enviamos el CRLF para indicarle que ha
                #finalizado la petición.
                self.target.sendall(self.CRLF)
                #Ahora nos toca recibir la respuesta y renviarla
                #al navegador.
                response_code = self.forward_response()
            else:
                allowed = "Blocked"
                response_code = 403
                #Si la petición ha sido bloqueada respondemos
                #al cliente con un mensaje de error
                self.send_error_message()
        finally:
            #Finalmente logueamos el resultado.
            log_line = "%s %s %s: %s://%s%s\n" % (self.command,
```

```
                                                    response_code,
                                                    allowed,
                                                    self.protocol,
                                                    host,
                                                    self.path)
            if VERBOSE: print log_line
            with open('log/VidiProxy.log', 'a') as log_file:
                log_file.write(log_line)

    #Definimos los métodos do_${HTTP_METHOD} para que llamen al
    #método genérico do_request()
    def do_GET (self):
        self.do_request()
    def do_POST (self):
        self.do_request()
```

7.3.2 Filtrado mediante listas

Aunque en un futuro el dispositivo debe conectarse al servicio de la nube para decidir si permite o no la conexión, para este proyecto nos basta con que pueda realizar el filtrado mediante listas. Aunque se debe dejar el código de tal forma que sea fácil de modificar para poder conectar al servicio en la nube.

Para ello definiremos los filtros como "plugins" que se puedan añadir o quitar y que sigan una interfaz para que a la hora de diseñar el proxy no nos tengamos que preocupar de como están implementados.

En el apartado anterior se vió cómo se definen qué listas se van a aplicar. En la clase ProxyHTTPHandler vimos como existía el atributo `self.filters`, que es una lista en la que añadimos una serie de objetos. Y luego hacemos una llamada al método `allowed`.

```
class ProxyHTTPHandler (BaseHTTPRequestHandler):
    (...)
    def __init__ (self, request, client_address, server):
        self.filters = [ WhiteListFilter(), BlackListFilter() ]
        (...)
    def do_request (self):
```

```
        (...)
        if self.allowed():
            (...)
```

El método `allowed` recorre la lista `self.filters`, llamando al método `evaluate` de cada objeto en la lista, pasándole la URL a la que queremos acceder. El método evaluate deberá retornar `True` si debemos permitir el acceso a la URL, `False` si debemos impedírselo o `None` si debemos consultar con el siguiente filtro.

```
def allowed (self):
    for x in self.filters:
        ret = x.evaluate(self)
        if ret is not None:
            return ret
    return True
```

De este modo podemos implementar las listas blancas y negras, pero dejamos una interfaz para poder crear una nueva clase que emplee filtros más complejos sin tener que realizar muchas modificaciones. La lista blanca y negra se implementa fácilemtne de la forma:

```
class WhiteListFilter (object):

    def __init__ (self, urls=["google.com"]):
        self.urls = urls

    def evaluate (self, request):
        for url in self.urls:
            if url in request.headers['host']:
                return True
        return None


class BlackListFilter (object):

    def __init__ (self, urls=['4chan.org']):
        self.urls = urls

    def evaluate (self, request):
        if request.headers['host'] in self.urls:
            return False
        return None
```

7.3.3 "Man In The Middle" en conexiones HTTPS

El proxy ha de ser capaz de inspeccionar el tráfico HTTP tanto cuando viaja en texto plano como cuando viaja a través de una conexión TLS/SSL, el problema aquí es que el objetivo de TLS/SSL es precisamente ese, evitar que terceros puedan inspeccionar las comunicaciones entre el servidor y el navegador.

Si decidimos simplemente que el navegador establezca un túnel TLS/SSL contra el proxy y que el proxy establezca su propio túnel hasta el servidor, el navegador se daría cuenta de ello dado que, o bien le entregaríamos un certificado no válido (generado por nosotros) o le entregaríamos el certificado de servidor pero los datos estarían cifrados con una clave privada distinta a la del certificado, en ambos casos el navegador le mostraría un mensaje al usuario indicando que la comunicación no es segura, lo cual sería bastante molesto para el usuario, así como bastante inseguro al enmascarar errores reales con la conexión TLS/SSL entre el proxy y el servidor.

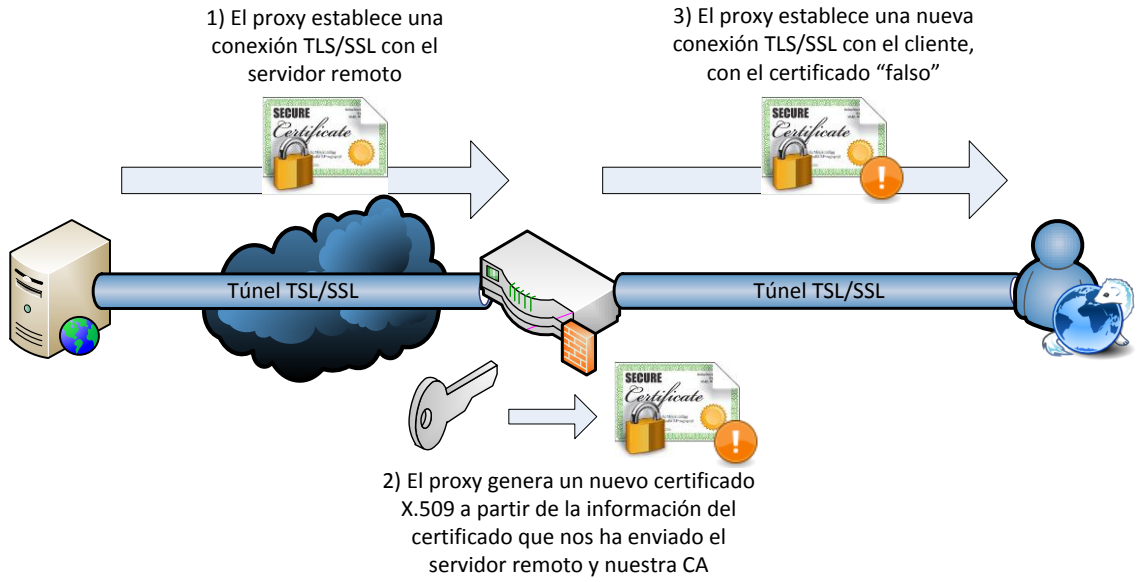
Para evitar todo esto, el proxy deberá ser capaz de generar un certificado válido para ese dominio con el que firmará el túnel TLS/SSL entre el proxy y el navegador. La única forma de generar un certificado válido es usar una CA que sea reconocida por el cliente como confiable. Dado que no podemos tener acceso (legalmente) a ninguna CA aceptada por la mayoría de clientes, la solución que nos queda es crear una CA propia y solicitar al cliente que configure el navegador para que considere esa CA como reconocida.

Para generar los certificados CA emplearemos OpenSSL, un toolkit de código abierto que implementa SSL y TLS así como una serie de bibliotecas criptográficas de propósito general. Dicho toolkit se emplea en bastantes escenarios a nivel empresarial.

Con el siguiente comando de OpenSSL podremos tener nuestra propia CA para firmar certificados:

```
$ openssl req -x509 -days ${DIAS} -newkey rsa:4096 -nodes\
    -keyout ${CAKEY} -out ${CACERT}
```

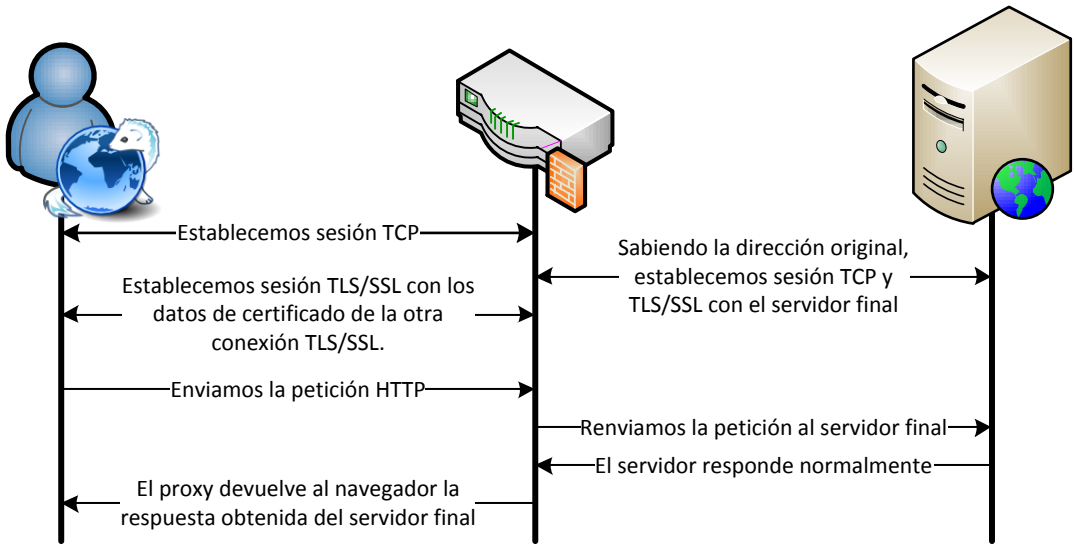
Donde `${DIAS}` es el numero de días que queremos que el certificado sea válido (a partir de hoy). `${CAKEY}` es la ruta donde queremos que se guarde la clave privada, que es lo que emplearemos para cifrar y firmar las conexiones. Y `${CACERT}` la ruta donde vamos a guardar el certificado público de la CA, que es el que tendremos que instalar en el navegador del cliente para que no muestre errores en las sesiones TLS/SSL.



Generación dinámica de un certificado X.509

Para generar los certificados TLS/SSL dinámicamente emplearemos M2Crypto, una biblioteca criptográfica para Python que se apoya en OpenSSL, aunque hace tiempo que no se actualiza y la documentación es bastante mejorable, cubre una gran cantidad de aspectos criptográficos que emplearemos para el proxy.

La idea de todo es: justo cuando realizamos el *handshake* TCP con el cliente, antes de gestionar el *handshake* de TLS/SSL, nos conectamos con el servidor remoto y establecemos el túnel TLS/SSL, a partir del certificado X.509 que nos entrega el certificado remoto generamos un nuevo certificado firmado con la CA que tenemos instalada en el cliente. Con ese certificado estableceremos el túnel TLS/SSL con el cliente.



Esquema de funcionamiento HTTPS de VidiProxy

Con el siguiente código obtenemos el certificado X.509 de un servidor remoto y generamos nuestro propio certificado, firmado por nuestra CA.

```
import ssl
import hashlib

from M2Crypto import SSL
```

```
from M2Crypto import X509
from M2Crypto import RSA
from M2Crypto import EVP

def forge_x509_cert (addr, cacert, cakey):
    '''Generamos un certificado X.509 para el dominio
    definido en addr.

    @param addr Dirección en la que encontraremos el
                certificado original.
    @param cacert M2Crypto.X509 Certificado de la CA.
    @param cakey M2Crypto.EVP Clave privada de la CA.
    @return Retorna el certificado y la clave privado`
            generados.'''

    #Obtenemos el certificado del servidor original.
    orig_pem = ssl.get_server_certificate(addr)
    orig_cert = X509.load_cert_string(orig_pem,
                                      format=X509.FORMAT_PEM)

    new_cert = X509.X509 ()
    #Copiamos los atributos del certificado original.
    new_cert.set_subject(orig_cert.get_subject())
    new_cert.set_version(orig_cert.get_version())
    new_cert.set_serial_number(
                  int(hashlib.md5(addr[0]).hexdigest(), 16))
    new_cert.set_not_before(orig_cert.get_not_before())
    new_cert.set_not_after(orig_cert.get_not_after())

    # En caso que exista la extensión X.509v3 subjectAltName
    #en el certificado original, también la copiamos en el
    #nuevo certificado.
    try:
        new_cert.add_ext(orig_cert.get_ext('subjectAltName'))
    except LookupError:
        pass

    # Copiamos el issuer de la CA en el certificado.
    new_cert.set_issuer(cacert.get_issuer())

    # Generamos la clave privada.
    key_pair = RSA.gen_key(1024, 0x1000 | 1)
    new_key = EVP.PKey(md='sha1')
    new_key.assign_rsa(key_pair)

    # Generamos la clave pública y firmamos el certificado
    new_cert.set_pubkey(new_key)
    new_cert.sign(pkey= cakey, md='sha1')

    return new_cert, new_key
```

Dado que este proceso es bastante costoso, sería interesante implementar una caché que guardara los certificados generados de modo que no tenga que volver a conectarse con el servidor remoto y generar las claves RSA de 1024 bits cada vez que nos volvamos a conectar e ese dominio.

De este modo cuando se recibe una conexión TCP, miraríamos si tenemos la dirección IP destino en la caché, si es así ya podríamos establecer la sesión TLS/SSL con el cliente usando el certificado que tenemos en la caché.

```
import threading

x509_cache = {}
rlock = trheading.RLock()

def get_cert (addr):
    '''Retorna el certificado y la clave privada para ese
    hostname.

    @param addr Tupla que consta en la dirección IP y el
                puerto al que nos conectamos
    @return Retorna el certificado y la clave privada para
            firmar la conexión TLS/SSL entre el proxy y el
            navegador'''

    # Definifmos el hostname de la forma IP:PORT
    hostname = addr[0] +':'+ str(addr[1])
    # Dado que el proxy emplea threads para gestiones
    #distintas conexiones de forma paralela debemos
    #asegurarnos que esta operación sea atómica.
    rlock.acquire()

    if hostname in self.x509_cache:
        # Si el certificado que necesitamos está en la caché,
        #lo servimos directamente.
        cert, key =  self.x509_cache[hostname]
    else:
        # En caso que el certificado no se encuentre en la
        #caché, lo generamos.
        cert, key = (addr, cacert, cakey)
        x509_cache[hostname] = (cert, key)

    rlock.release()
    return cert, key
```

En cuanto a la clase encargada de gestionar la conexión será exactamente igual a la que empleamos para el HTTP solo que en el `__init__` "envolveremos" el socket con la capa de TLS/SSL.

```
from socket import AF_INET, SOCK_STREAM
from socket import socket
import ssl
from BaseHTTPServer import BaseHTTPRequestHandler


class ProxyHTTPSHandler (ProxyHTTPHandler):

    def __init__ (self, request, client_address, server):
       '''Establecemos el handler de la conexión TLS/SSL
       envolviendo los sockets con la clase ssl de Python '''

        # Creamos los sockets como en el handler de HTTP
        self.server = server
        self.request = request
        addr = self.get_original_address()
        self.protocol = 'https'
        self.target = socket(AF_INET,SOCK_STREAM)
        self.target.connect(addr)
        self.target.settimeout(2)
        # Establecemos la conexión TLS con el servidor final
        ssl.wrap_socket(self.target,
                        ssl_version=ssl.PROTOCOL_SSLv23,
                        cert_reqs=ssl.CERT_NONE,
                        ca_certs='trusted_certs/certs.pem')
        # Obtenemos el certificado de la caché, si existe y si
        #no lo creamos
        cert, key = self.server.x509_cache.get_cert(addr)
        # Establecemos una sesión TLS/SSL sobre la conexión
        #entre el navegador y el proxy
        ssl.wrap_socket(request, key=key, cert=cert,
                        server_side=True)
        self.filters = [ WhiteListFilter(), BlackListFilter() ]
        BaseHTTPRequestHandler.__init__(self, request,
                                        client_address,
                                        server)
```

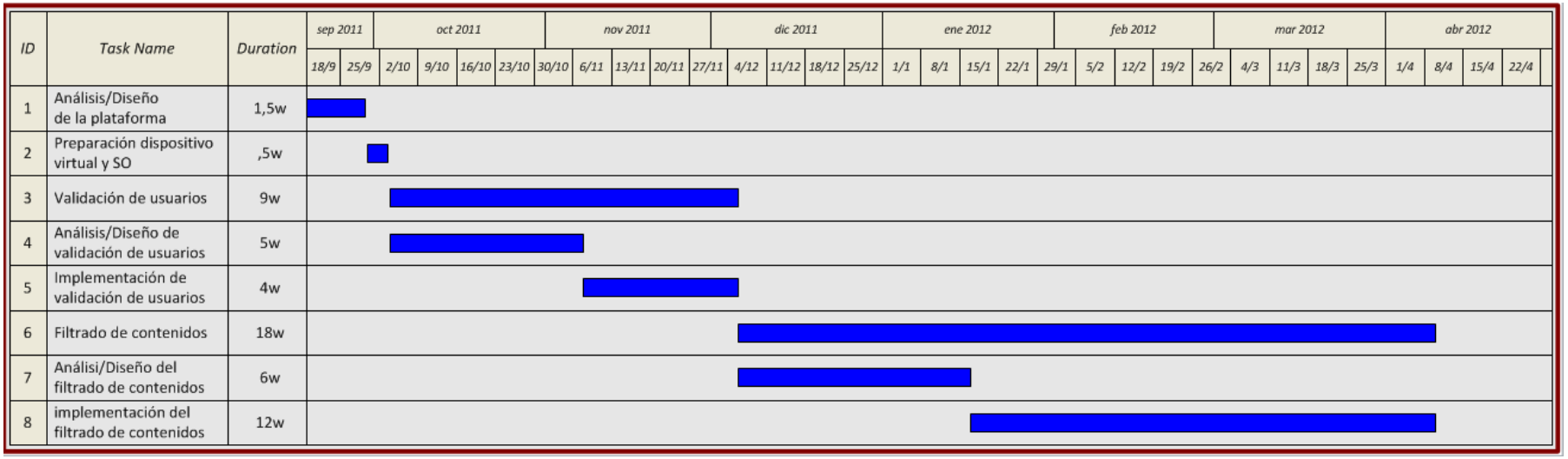
8 Estudio económico

En este capítulo se analizará el impacto económico que representaría este proyecto en una empresa.

Dado que para el proyecto no se ha empleado software con licencias ni ningún tipo de hardware específico, se estimará el coste económico a partir de las horas empleadas.

Finalmente, debido a que durante el desarrollo del proycto se tuvo que cambiar el planteamiento e implementar desde cero las soluciones de validación de usuarios y de filtrado de contenidos, ambas tareas se alargaron considerablemente debido a tener que diseñarlas e implementarlas desde cero:

A continuación mostramos el diagramad e Gantt con el tiempo real de desarrollo del proyecto:


ID
Task Name
Duration
sep 2011
oct 2011
nov 2011
dic 2011
ene 2012
feb 2012
mar 2012
abr 2012
18/9
25/9
2/10
9/10
16/10
23/10
30/10
6/11
13/11
20/11
27/11
4/12
11/12
18/12
25/12
1/1
8/1
15/1
22/1
29/1
5/2
12/2
19/2
26/2
4/3
11/3
18/3
25/3
1/4
8/4
15/4
22/4
1 Análisis/Diseño de la plataforma 1,5w
2 Preparación dispositivo virtual y SO ,5w
3 Validación de usuarios 9w
4 Análisis/Diseño de validación de usuarios 5w
5 Implementación de validación de usuarios 4w
6 Filtrado de contenidos 18w
7 Análisi/Diseño del filtrado de contenidos 6w
8 implementación del filtrado de contenidos 12w

Para el estudio se tendrán en cuenta dos perfiles: analista/arquitecto de sistemas y programador/técnico de sistemas.

Para el perfil de analista/arquitecto de sistemas se asignan las siguientes tareas:

Tareas	Horas
Ánálisis/diseño de la solución.	30h
Ánálisis/diseño de la validación de usuarios.	100h
Ánálisis/diseño del filtrado de contenidos.	120h
Total	**250h**

Para el perfil programador/técnico de sistemas se le asignan las siguientes tareas:

Tareas	Horas
Preparación del dispositivo virtual y el sistema operativo.	10h
Implementación de la validación de usuarios.	80h
Implementación del filtrado de contenidos	480h
Total	**570h**

Si se asume que el precio por hora del perfil de analista/arquitecto de sistemas es de 40€, y que el precio por hora del perfil de programador/técnico de sistemas es de 20€, el coste del proyecto sería:

Perfil	Precio	Horas	Total
Analista/arquitecto de sistemas	40€/hora	250h	10000€
Programador/técnico de sistemas	20€/hora	570h	11400€
Total			**21400€**

9 Conclusiones

El proyecto se ha alejado de la idea original, que era empaquetar y configurar diferentes piezas de software para que trabajaran juntas y finalmente se ha tenido que implementar desde cero las dos aplicaciones principales. Cosa que ha permitido diseñar esas piezas de software específicamente para el proyecto, lo que permitirá una mejor adaptación a necesidades comerciales en un futuro. Incluso con esa desviación, los objetivos del proyecto se han cumplido.

De hecho, gracias a esa desviación me he visto obligado a aprender el funcionamiento de los protocolos HTTP y TLS/SSL con el detalle necesario para poder implementar el VidiProxy, lo cual considero que ha hecho mucho más provechoso el proyecto.

10 Líneas de trabajo futuro

Se ha finalizado el desarrollo de la maqueta y ahora cumple con las funcionalidades definidas en los objetivos del proyecto, aunque aún se puede seguir trabajando para conseguir mejorar el rendimiento, así como añadir algunas nuevas funcionalidades:

- **Rescribir el proxy a C**, aunque Python es un lenguaje muy práctico y rápido de escribir, no es demasiado eficiente en comparación con C debido al "overhead" del intérprete. Debido a que Python y sus bibliotecas están escritos en C, es relativamente sencillo ir rescribiendo partes del código en C y enlazarlas con el código escrito en Python.
- **Implementar extensiones TLS como (Server Name Indication, SNI**), SNI es una de las extensiones a TLS definidas en el RFC 6066, que permite servir diversos certificados X.509 desde una misma IP. Para ello el cliente debe enviar el nombre del dominio virual al que se se quiere conectar. Aunque todavía no está muy extendido, ya hay algunos sitios en los que se emplea como, por ejemplo, GMail.
- **Integrar Chell con VidiProxy**, debido a que se decidió implementar nuestro propio portal cautivo antes de decidir qué hacer con el proxy, se implementó como un CGI corriendo sobre un servidor web independiente. Pero visto que también hemos tenido que implementar el proxy, tal vez sea buena idea integrar el portal cautivo dentro de VidiProxy.
- **Optimizar el sistema operativo**, para el alcance del proyecto hemos empleado una Debian GNU/Linux con la configuración por defecto, pero

cuando despleguemos el proxy dentro de un dispositivo será necesario realizar optimizaciones a la configuración para que se ejecute sin problemas en un hardware limitado.

11 Bibliografía

11.1 SSL/TLS

- J. Traver, *Tutorial OpenSSL/X509*

  http://spi1.nisu.org/recop/al01/pepe/

- D. Koren, *TLS (Transport Layer Security) and SSL (Secure Socket Layer)*

  http://www3.rad.com/networks/applications/secure/tls.htm

- T. Dierks, E. Rescorla, *The Transport Layer Security (TLS) Protocol*

  http://tools.ietf.org/html/rfc5246

- D. Eastlake, *Transport Layer Security (TLS) Extensions: Extension Definitions*

  http://tools.ietf.org/html/rfc6066

- Microsoft, *CA Certificates Technical Reference*, 2003

  http://technet.microsoft.com/en-us/library/cc736984

11.2 HTTP/HTTPS

- R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter. Leach, T. Berners-Lee, *Hypertext Transfer Protocol*, 1999 http://tools.ietf.org/html/rfc2616
- E. Rescorla, *HTTP Over TLS,* 2000 http://tools.ietf.org/html/rfc2818
- B. Krisgbanurthy, J. Mogul, D. Kristol, *Key Differences between HTTP/1.0 and HTTP/1.1*, 1999 http://www8.org/w8-papers/5c-protocols/key/key.html

11.3 Python

- Python Software Foundation, *Python Documentation* http://docs.python.org
- M. Pilgrim, *Dive into Python 3* Ed. Apress, 2009

- N. Gift, J. Jones, *Python for Unix and Linux System Administrators* Ed. O'Reilly, 2008

11.4 Otro

- F. Márquez, *UNIX Programación avanzada*, Ed. Ra-Ma, 2004
- A. Rousskov, M. Kool, A. Jeffries, *Dynamic SSL Certificate Generation*, 2008 – 2012. http://wiki.squid-cache.org/Features/DynamicSslCert

12 Anexos

12.1 Validación de usuarios del portal cautivo

A continuación se adjunta el código del CGI encargado de comprobar que los credenciales introducidos por el usuario en el portal cautivo son correcto y, si es el caso, generar las reglas que permitirán a ese usuario navegar a través del proxy.

```
import os
import sys
import time
import urllib
import hashlib
import cgi

class CgiValidacion:

    def __init__(self):
        self.cgi_form = cgi.FieldStorage()
        self.login = {}
        with open('/path/to/chell.passwd', 'r') as passwd_file:
            for line in passwd_file.readlines():
                line = line.split(':')
                self.login[line[0]] = line[1]

    def validar (self, usr, paswd):
        '''Validamos si el usuario y password introducidos
        existen en el sistema

        @params usr El usuario que quermos validar.
        @params paswd La contraseña introducida por el usuario.'''
        digest = hashlib.sha256(paswd).hexdigest()
        if usr in self.login and self.login[usr] == digest:
            return True
        else:
            return False

    def redirect (self, uri):
        print 'HTTP/1.1 302 Found'
        print 'Location: %s' % uri
        print
        sys.exit()

    def cgi_get_atribute (self, var_name):
        if var_name in self.cgi_form:
            var = self.cgi_form[var_name].value
        else:
            var = ''
        return var
```

```
    def cgi_get_environ (self, var_name):
        if var_name in os.environ::
            var = os.environ[var_name]
        else:
            var = ''
        return var

    def run (self):
        user = self.cgi_get_atribute('user')
        password = self.cgi_get_atribute('password')
        dst_url = self.cgi_get_atribute('url')

        if not user or not password:
            self.redirect('/login.py?url=%s' % urllib.quote(dst_url))
            print "dst_url: %s<br>urllib.quote(dst_url): %s" %
                                        (dst_url, urllib.quote(dst_url))
        else:
            if self.validar(user, password):
                if dst_url:
                    remote_ip = self.cgi_get_environ('REMOTE_ADDR')
                    command = "/usr/bin/sudo /usr/sbin/conntrack"
                    command += "-s %s -D  2>&1 > /dev/null"% remote_ip
                    os.system(command)
                    command = "/usr/bin/sudo /sbin/iptables
                    command += "-t nat -I PREROUTING "
                    command += "-s %s -d 0.0.0.0/0 -p tcp --dport 80"%
                                                              remote_ip
                    command += "-j REDIRECT --to-port 3128"
                    os.system(command)
                    self.redirect('http://%s' % dst_url)
                else:
                    self.redirect('/login.py')
            else:
                self.redirect('/login.py?err=1&url=%s' %
                                                 urllib.quote(dst_url))

CgiValidacion().run()
```

ENGINYERIA TÈCNICA EN INFORMÀTICA DE SISTEMES

Treball fi de carrera presentat el dia de de 200

a la Facultat de Matemàtiques de la Universitat de Barcelona,

amb el següent tribunal:

Dr.	President
Dr.	Vocal 1
Dr.	Secretari

Amb la qualificació de: