Universidad de Murcia

Servicios Telemáticos

PCEO

Entrega de Prácticas de Servicios Telemáticos

Autor:
Jose Antonio Lorencio Abril
49196915F
joseantonio.lorencioa@um.es

Profesor:
Esteban Belmonte Rodríguez

Junio 2020

Índice

Índice de cuadros

Índice de figuras

1. Introduccion

En esta memoria presento el trabajo que he realizado para llevar a cabo la práctica del de la asignatura Servicios Telemáticos.

Explicaré las decisiones de implementación tomadas, así como los problemas que han surgido en el desarrollo de la práctica. También incluiré trazas de Wireshark que muestren el correcto funcionamiento de mi escenario e incluiré código o cualquier otro tipo de tabla/imagen cuando lo estime oportuno para la correcta comprensión del contenido expuesto. Terminaré el documento con mi valoración personal sobre la práctica.

2. Descripción del escenario desarrollado y versiones de Software

Desde que comenzamos la práctica, nuestro objetivo ha sido desarrollar el siguiente escenario:

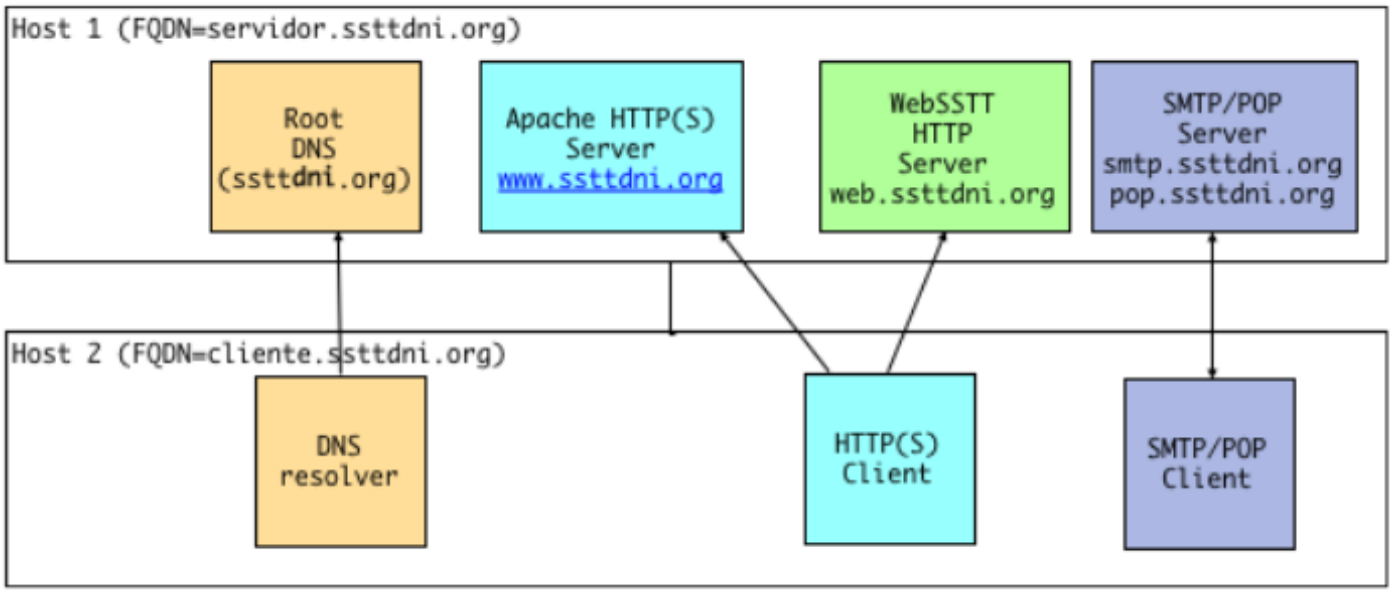


Figura 1: Escenario de la práctica

Como vemos, consiste de dos hosts, uno que consideraremos servidor (servidor.sstt6915.org) con IP 192.168.56.104, y otro que consideraremos cliente (cliente.sstt6915.org) con IP 192.168.56.101.

Para llevarlo a cabo, hemos simulado ambos host en dos máquinas virtuales dentro de una misma máquina anfitrión.

El servidor ofrece al cliente los siguientes servicios:

- Servicio DNS
- Apache HTTP y HTTPs
- Servidor de correo POP, SMTP
- Conexión dando uso a IPsec

Por su lado, el cliente deberá ser configurado para pedir estos servicios correctamente al servidor. Estos detalles se analizarán detenidamente a lo largo del presente documento.

3. Descripción de la configuración de los servicios

3.1. DNS

El servicio DNS va a ser implementado usando la herramienta Bind9, instalada en el servidor y configurada de manera conveniente.

Para ello, vamos a seguir los siguientes pasos:

1. Instalamos Bind:

```
1        sudo apt-get install bind9
2
```

Cuadro 1: Instalando Bind9

2. Una vez instalado, lo iniciamos:

```
1        sudo /etc/init.d/bind9 start
2
```

Cuadro 2: Instalando Bind9

3. Ahora debemos modificar los ficheros de configuración que usa Bind para su funcionamiento. Concretamente, estos ficheros son */etc/bind/named.conf.options* y */etc/bind/named.conf.local*. En el primero, como queremos que reenvíe al servidor de la UM las peticiones que no sepamos responder satisfactoriamente, añadiremos el servidor DNS de la UM en forwarders:

```
1        forwarders {
2            155.54.1.1;
3        };
4        ...
5        dnssec-validation no;
6
```

Cuadro 3: Fragmento de */etc/bind/named.conf.options*

También ponemos dnssec-validation a 'no', pues no implementamos el servicio DNS con seguridad. En */etc/bind/named.conf.local* debemos añadir nuestra zona, de la siguiente forma:

```
1        zone sstt6915.org IN {
2
3            type master;
4            file "/etc/bind/db.sstt6915.org.zone"
5        };
6
```

Cuadro 4: Fragmento de */etc/bind/named.conf.local*

En este fragmento podemos ver que la zona se llama *sstt6915.org*, como se nos requiere en el enunciado. El 'type master' indica que nuestro servidor tiene la autoridad para esta zona. La última línea índica dónde se encuentra el fichero de zona, que contiene la información sobre nuestro dominio. Por supuesto, este fichero debemos crearlo y escribirlo correctamente.

4. Creamos el fichero */etc/bind/db.sstt6915.org.zone*, que debe quedar como sigue:

```
;
; BIND data file for local loopback interface
;
$TTL    604800
$ORIGIN sstt6915.org
@   IN  SOA sstt6915.org. root.sstt6915.org. (
                    2           ;   Serial
                    604800      ;   Refresh
                    86400       ;   Retry
                    2419200     ;   Expire
                    604800  )   ;   Negative Cache TTL
;
@   IN  NS  dns.sstt6915.org.
dns IN  A   192.168.56.104
web IN  A   192.168.56.104
www IN  A   192.168.56.104
smtp    IN  A   192.168.56.104
pop IN  A   192.168.56.104
@   IN  MX  10  smtp

```

Cuadro 5: */etc/bind/db.sstt6915.org.zone*

Y aquí hay varias cosas que comentar:

- $TTL: indica el tiempo durante el que los Resource Records (RR) de nuestra zona son válidos.
- $ORIGIN: se usa como nombre común, con el que se completarán los nombres no cualificados. Es decir, los nombres que no terminen con un punto ('.') serán completados añadiento, al final, *sstt6915.org*.
- Después vemos los registros de recursos:
  - SOA: contiene información general aplicable a todos los registros de la zona.
  - NS: identifica el servidor de nombres de nuestro dominio.
  - A: con este tipo de registros asignamos a las direcciones indicadas los nombres que se ven.
  - MX: este registro se utiliza para encaminar el correo electrónico. Es el encargado de indicar qué servidor se encarga de llevar el email en nuestro dominio. La importancia de este registro se detallará en la explicación de SMTP/POP, donde se nos pregunta explícitamente sobre esta cuestión.

En este punto, el servidor ya estaría dispuesto para dar el servicio DNS, pero debemos hacer una pequeña modificación del cliente.

El cliente, cuando desconoce la IP asociada a un host del que requerimos información, se dirige al fichero */etc/resolv.conf*, donde se encuentra la IP a la que debe hacer la petición DNS. Por tanto, debemos modificar este fichero para que sea nuestro servidor el preguntado. El problema es que este fichero se modifica automáticamente al reiniciar la máquina, por lo que debemos hacer una pequeña jugada para no tener que andar modificándolo cada vez que encendamos el cliente. Los pasos a seguir son los siguientes:

1. Nos aseguramos que el contenido del archivo es que queremos conservar, en mi caso, el fichero queda así:

```
#...
nameserver 192.168.56.104

```

Cuadro 6: */etc/resolv.conf*

Y hacemos una copia del archivo:

```
sudo su
cp /etc/resolv.conf /etc/resolv.conf.bak

```

Cuadro 7: Copiando */etc/resolv.conf*

2. Hacemos ahora que este nuevo archivo sea inalterable:

```
chattr +i /etc/resolv.conf.bak

```

Cuadro 8: Haciendo */etc/resolv.conf.bak* inalterable

3. Eliminamos el antiguo archivo:

```
rm /etc/resolv.conf

```

Cuadro 9: Eliminando el */etc/resolv.conf* antiguo

4. Por último, renombramos el archivo creado en el paso 1, para que sea este el archivo donde el cliente buscará el servidor DNS al que preguntará. Ahora no se modiciará, ya que lo hemos hecho inalterable.

```
cp /etc/resolv.conf.bak /etc/resolv.conf

```

Cuadro 10: Renombrando */etc/resolv.conf.bak* para que funcione

Ya debería funcionar el servicio DNS perfectamente. Podemos probarlo usando el comando dig.

```
alumno@desktop:~$ dig www.sstt6915.org

; <<>> DiG 9.10.3-P4-Ubuntu <<>> www.sstt6915.org
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 566
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 2

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;www.sstt6915.org.              IN      A

;; ANSWER SECTION:
www.sstt6915.org.       604800  IN      A       192.168.56.104

;; AUTHORITY SECTION:
sstt6915.org.           604800  IN      NS      dns.sstt6915.org.

;; ADDITIONAL SECTION:
dns.sstt6915.org.       604800  IN      A       192.168.56.104

;; Query time: 0 msec
;; SERVER: 192.168.56.104#53(192.168.56.104)
;; WHEN: Fri May 22 10:33:51 CEST 2020
;; MSG SIZE  rcvd: 95
```

Figura 2: Probando DNS con dig

Y vemos que, efectivamente, el funcionamiento es correcto.

3.2. SMTP/POP

3.2.1. Desplegando SMTP

Lo primero que hacemos es instalar la herramienta Exim4:

```
1    sudo apt-get update
2    sudo apt-get install exim4
3    sudo dpkg-reconfigure exim4-config
```

Cuadro 11: Instalando *Exim4*

El último comando sirve para entrar en la configuración de esta herramienta. Esta configuración queda así:

```
Tipo general del servidor: Internet site (la primera opción)
Nombre del sistema de correo: sstt6915.org
Direcciones IP en las que recibir conexiones SMTP (en blanco, cualquier IP);
Destinos de los que se acepta correo: sstt6915.org
Dominio para que se puede reenviar correo: sstt6915.org; um.es
Máquinas para las cuales reenviar correo: 192.168.56.0/24 Limitar consultas DNS: NO
Formato de buzón de correo: Maildir
Dividir ficheros de configuración: NO
```

Cuadro 12: Configurando *Exim4*

3.2.2. Desplegando POP

En esta ocasión, vamos a usar *dovecot*. Para instalarlo:

```
sudo apt-get install dovecot-pop3d
```

Cuadro 13: Instalando *dovecot*

Como queremos que la autenticación sea débil y basada en texto plano, debemos configurarlo. Ya que, por defecto esto no se permite. Para ello, debemos modificar, en el fichero */etc/dovecot/conf.d/10-auth.conf*, las líneas:

```
...
disable_plaintext_auth = no
...
auth_mechanisms = plain
...
!include auth-system.conf.ext
...
```

Cuadro 14: Fragmento de */etc/dovecot/conf.d/10-auth.conf*

Y debemos modificar, también, el fichero */etc/dovecot/conf.d/10-mail.conf*, especificando el formato de los buzones. Para ello, solo hay que modificar una línea de este fichero:

```
...
mail_location= maildir:~/Maildir
...
```

Cuadro 15: Fragmento de */etc/dovecot/conf.d/10-mail.conf*

Ahora procedemos a crear los usuarios de correo de nuestro escenario:

```
useradd nombre1\_6915 -m
passwd nombre1\_6915
introduzca la nueva contraseña de UNIX: alumno
vuelva a introducir la constraseña de UNIX: alumno
```

Cuadro 16: Creando los usuarios de correo

Y repetimos esto para crear otro usuario *nombre2_ 6915*

Ahora comprobamos si en su directorio se ha creado un directorio Maildir. De no ser así, nos logeamos como estos usuarios y lo creamos a mano:

```
maildirmake.dovecot ./Maildir
```

Cuadro 17: Creando los buzones de cada usuario

Y en este punto el servidor ya está preparado para enviar y recibir correo. Podemos hacer una prueba mediante línea de comandos, desde el cliente:

```
alumno@desktop:~$ telnet smtp.sstt6915.org 25
Trying 192.168.56.104...
Connected to smtp.sstt6915.org.
Escape character is '^]'.
220 server ESMTP Exim 4.86_2 Ubuntu Fri, 22 May 2020 11:17:16 +0200
HELO sstt6915.org
250 server Hello sstt6915.org [192.168.56.101]
MAIL FROM: nombre1_6915@sstt6915.org
250 OK
RCPT TO: nombre2_6915@sstt6915.org
250 Accepted
DATA
354 Enter message, ending with "." on a line by itself
Subject: Buenos días
From: nombre1_6915@sstt6915.org
To: nombre2_6915@sstt6915.org

Esto es una prueba.

.
250 OK id=1jc3oa-0003yT-0G
quit
221 server closing connection
Connection closed by foreign host.
```

Figura 3: Enviando un correo con SMTP mediante telnet

Y ahora, vamos a usar Pop para recuperar este correo:

```
alumno@desktop:~$ telnet pop.sstt6915.org 110
Trying 192.168.56.104...
Connected to pop.sstt6915.org.
Escape character is '^]'.
+OK Dovecot ready.
user nombre2_6915@sstt6915.org
+OK
pass alumno
+OK Logged in.
list
+OK 4 messages:
1 978
2 1227
3 998
4 543
.
retr 4
+OK 543 octets
Return-path: <nombre1_6915@sstt6915.org>
Envelope-to: nombre2_6915@sstt6915.org
Delivery-date: Fri, 22 May 2020 11:18:47 +0200
Received: from [192.168.56.101] (helo=sstt6915.org)
        by server with smtp (Exim 4.86_2)
        (envelope-from <nombre1_6915@sstt6915.org>)
        id 1jc3oa-0003yT-0G
        for nombre2_6915@sstt6915.org; Fri, 22 May 2020 11:18:47 +0200
Subject: Buenos días
From: nombre1_6915@sstt6915.org
To: nombre2_6915@sstt6915.org
Message-Id: <E1jc3oa-0003yT-0G@server>
Date: Fri, 22 May 2020 11:18:39 +0200

Esto es una prueba.

.
```

Figura 4: Recibiendo el correo usando POP mediante telnet

Vemos más correos, son de otras pruebas que realicé con anterioridad. Hemos comprobado, de esta forma, que el servicio de correo está configurado correctamente.

Ahora, para poder tener estos usuarios de correo funcionales en Thunderbird en el cliente, debemos configurar este. Este punto es bastante simple y no creo necesario explicar todos los pasos a seguir, ya que la interfaz de Thunderbird ayuda a saber qué debemos hacer. Solo notar que querremos configurar el servidor de salida con SMTP en el puerto 25, y el servidor de recepción con POP, en el puerto 110.

3.2.3. Se deberá analizar la relación entre el servicio SMTP y el DNS a través del registro MX. Indica si es necesario o no su uso en la práctica y por qué.

La relación entre estos dos servicios ha sido mencionada en el apartado de DNS. Como he señalado antes, el registro MX se encarga de indicar qué servidor es el encargado de manejar el email en nuestro dominio. En nuestra práctica, realmente podríamos haber prescindido de su uso, porque se utiliza para hacer conocer a otros servidores de correo dónde enviarnos el correo. Sin embargo, nosotros enviamos y recibimos correos desde nuestro servidor. Nótese que en una situación real sí que sería necesario, pues podríamos recibir correo de servidores que no son el nuestro.

3.3. APACHE HTTP/HTTPS

3.3.1. HTTP

Para el despliegue del servicio HTTP vamos a usar el software de Apache:

```
1    sudo apt-get update
2    sudo apt-get install apache2
```

Cuadro 18: Instalando Apache

Las páginas a las que el servidor va a proporcionar acceso son realmente simples. Como se pedía que hubiera al menos una página referenciada, he usado dos archivos HTML. Uno de ellos consiste en un título, el logo de la UM, y un enlace a la otra página. Esta segunda página consta únicamente de un título y el logo de la UM. Las he realizado simples porque el objetivo de la práctica no era crear páginas web.

Pasamos ahora a la configuración del servicio. Lo primero que vamos a hacer es crear el Virtual Host. Para ello, en */etc/apache2/sites-available* creamos el fichero *sstt6915.conf*, con el siguiente contenido:

```
1    <VirtualHost *:80>
2        ServerAdmin alumno@sstt6915.org
3        ServerName www.sstt6915.org
4        DocumentRoot /var/www/sstt6915
5        <Directory /var/www/sstt>
6            Allow Override AuthConfig
7            AuthType Basic
8            AuthName "Acceso restringido a jose"
9            AuthBasicProvider file
10           AuhtUserFile    /etc/apache2/passwords
11           Require user jose
12           Order allow,deny
13           allow from all
14       </Directory>
15   </VirtualHost>
```

Cuadro 19: Archivo *sstt6915.conf*

Que queda configurado para que el servidor nos pida nombre de usuario y contraseña (respectivamente, 'jose' y 'jose'). Para crear esta contraseña, hacemos lo siguiente:

```
1    htpasswd-c /etc/apache2/passwords jose
2    New password: jose
3    Re-typenew password: jose
4    Adding password for user jose
```

Cuadro 20: Creando usuario de acceso a HTTP

Antes de activar el Virtual Host, creamos el fichero */var/www/sstt6915* e introducimos en su interior los archivos de los que requiera el servicio web. En mi caso, los dos archivos HTML y el logo de la UM en JPG.

Ahora solo resta activar el Virtual Host y reiniciar el servicio:

```
1    sudo a2ensite sstt6915.conf
2    service apache2 reload
3    service apache2 restart
```

Cuadro 21: Activando el VH y reiniciando Apache2

Para probarlo, basta usar Firefox en el cliente y conectarnos a *www.sstt6915.org*. Nos pedirá usuario y contraseña y, tras introducirlos, se nos mostrará el contenido de la página.

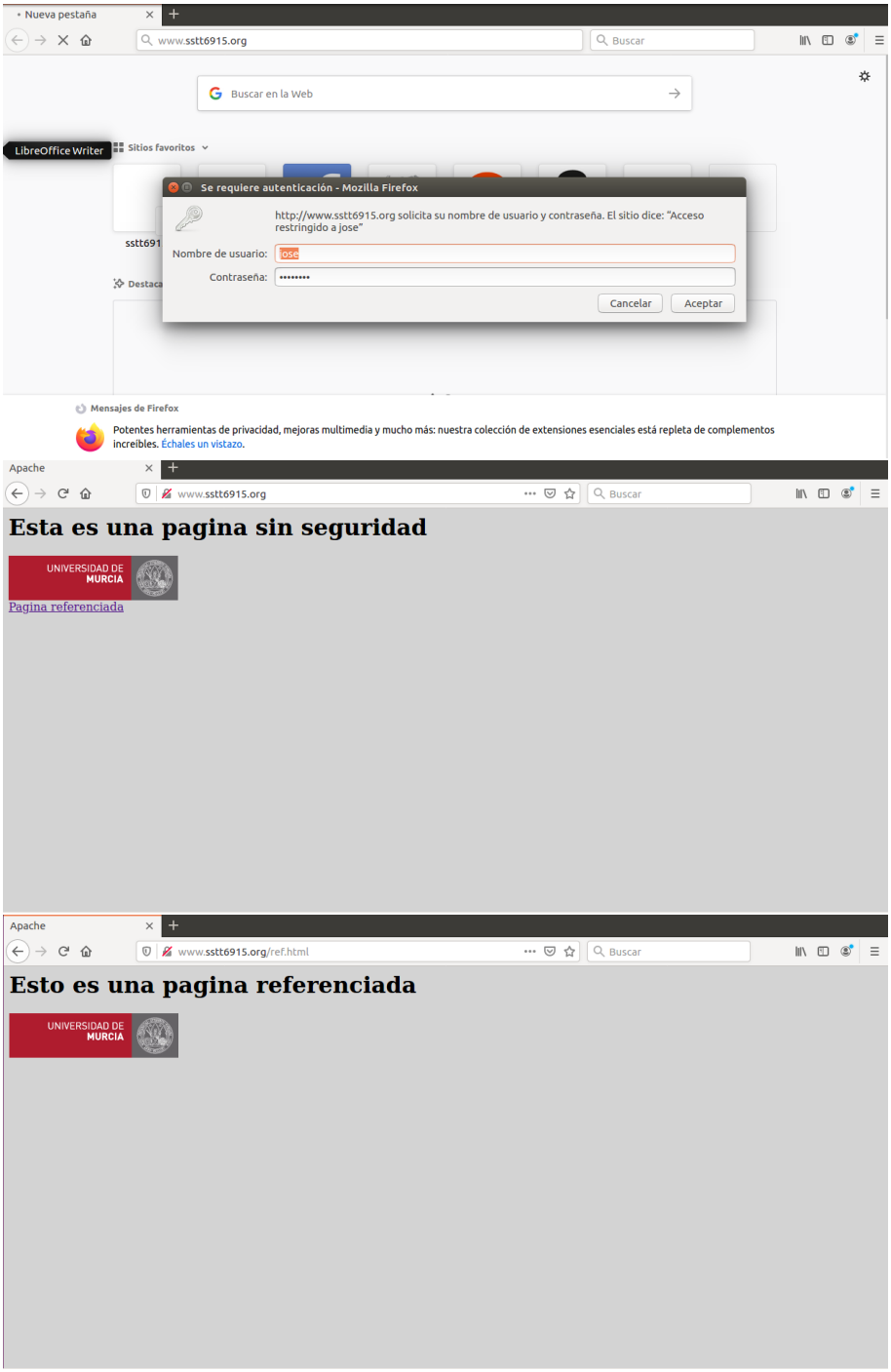


Figura 5: Accediendo al servicio Apache HTTP

Y funciona perfectamente. Pasamos ahora a HTTPS.

3.3.2. HTTPS

Para llevar esto a cabo, he seguido los pasos de las diapositivas de las prácticas 5 y 6. Lo primero que hacemos es preparar el entorno para PKI, que constará de una Autoridad de Certificación (CA) desplegada en el servidor. Para ello, ejecutamos, desde */home/alumno*:

```
    mkdir demoCA
    cd demoCA
    # (En ~/demoCA)
    echo 0 > crlnumber
    touch index.txt
    echo 01 > serial
    mkdir newcerts
    mkdir certs
    mkdir private
```

Cuadro 22: Creando la CA

Ahora vamos a */usr/lib/ssl/openssl.cnf* y en la sección *[CA_Default]*, modificamos:

```
    ...
    dir=/home/alumno/demoCA
    ...
    countryName_default =    ES
    ...
    stateOrProvince_default =    Murcia
    ...
    0.organizationName_default  =   UMU
    ...
    organizationalUnitName_default  =   joseantonio_6915
    ...
```

Cuadro 23: Fragmento de */usr/lib/ssl/openssl.cnf*

En 'dir' le indicamos a ssl dónde están los certificados. Las demás líneas definen la estructura de nombres.

Pasamos ahora a generar el certificado de la CA. Para ello, nos vamos al directorio */home/alumno/demoCA* y ejecutamos:

```
    openssl req -x509 -newkey rsa:2048 -keyout cakey.pem -out \
cacert.pem -days 3650
```

Cuadro 24: Creando el certificado de la CA

Se nos va requiriendo alguna información por terminal. Yo he introducido:

- Como contraseña he establecido 'jose'
- Los demás valores los he dejado como estaban por defecto. Excepto el CN=ca.sstt6915.org y el emailAddress=joseantonio.lorenci

Ahora tenemos generados el certificado de la CA (*cacert.pem*) y la clave privada (*cakey.pem*). Esta última la moveremos a */home/alumno/demoCA/private*.

Ahora nos disponemos a certificar el servicio web Apache. Permaneciendo en */home/alumno/demoCA*, para generar la solicitud de la ceritificación, así como la clave privada RSA, debemos ejecutar:

```
openssl req -new -nodes -newkey rsa:2048 -keyout serverkey.pem -out servercsr.pem
```

Cuadro 25: Creando el certificado del servidor Apache I

Lo dejo todo tal cual, excepto el CN=www.sstt6915.org. La contraseña que pide la dejo en blanco. Y, ahora, ejecutamos:

```
openssl ca -keyfile private/cakey.pem -in servercsr.pem -out servercert.pem -days 400
```

Cuadro 26: Creando el certificado del servidor Apache II

Se nos requerirá la contraseña de la clave privada de la CA y, tras introducirla, se ahbrá creado el certificado *servercert.pem*. Podemos visionar parte de su contenido:

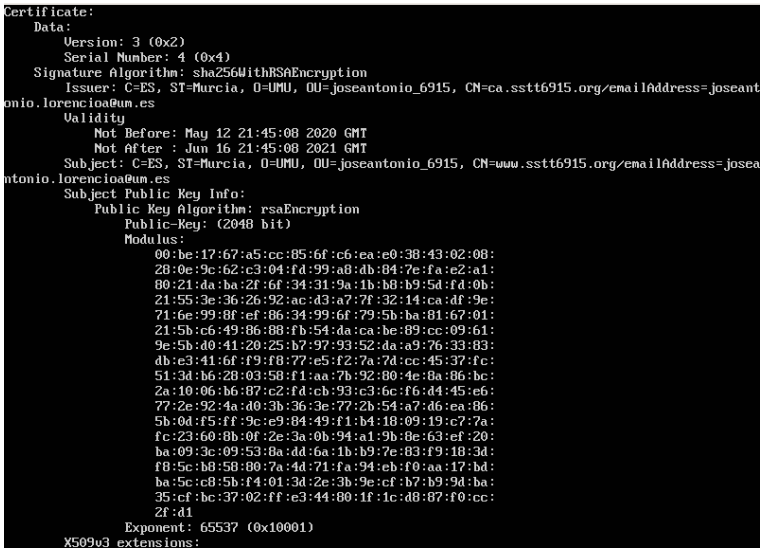


Figura 6: Certificado del Servidor Apache

Donde podemos observar la información de la CA (Issuer) y del servidor (Subject). Lo que resta del certificado no aporta nada de información.

Además, se ha creado una copia del certificado en */home/alumno/demoCA/newCerts*. Ya podemos eliminar el fichero *servercsr.csr*, pues ya no es necesario.

Pasamos ahora a proveer al cliente de un certificado propio para que la autenticación con certificados pueda ser satisfactoria. Para ello, seguimos los mismos pasos que hemos realizado para certificar el servidor. Al terminar, importamos el certificado y la clave privada al navegador, para lo que debemos ejecutar:

```
openssl pkcs12 -export -in clientcert.pem -certfile cacert.pem -inkey clientkey.pem -out clientcert.pfx
```

Cuadro 27: Importando el certificado y clave privada del cliente para el navegador

Generando un archivo *clientcert.pfx*, que contiene el certificado x509 del cliente y de la CA, así como la clave privada del cliente. Ahora debemos hacer llegar este certificado al cliente, esto podemos hacerlo usando el comando scp.

Desde el cliente, iremos a la configuración avanzada de Firefox, a la sección de *Certificados->Ver Certificados*, e importamos el certificado.

Solo resta hacer que Apache dé uso de los certificados para proporcionar el servicio HTTPS. Para ello, lo primero que hacemos es habilitar el módulo ssl de Apache:

```
sudo a2enmod ssl
```

Cuadro 28: Habilitando ssl

Y procedemos a crear un Virtual Host de forma similar, aunque no idéntica, a como lo hicimos para el servicio HTTP. Esta vez el Virtual Host solo constará de una única página HTML con título y el logo de la UM, que estarán en */var/www/sstt6915_ssl*. Debemos crear este último directorio y transferir los archivos ahí, así como, en */etc/apache2/sites-available*, crear *sstt6915_ssl.conf*, que quedará como sigue:

```
<IfModule mod_ssl.c>
    <VirtualHost *:443>
        ServerAdmin alumno@sstt6915.org
        ServerName www.sstt6915.org
        DocumentRoot /var/www/sstt6915_ssl

        SSLEngine on
        SSLCertificateKeyFile /home/alumno/demoCA/serverkey.pem
        SSLCertificateFile /home/alumno/demoCA/servercert.pem
        SSLCACertificateFile /home/alumno/demoCA/cacert.pem
        SSLVerifyClient require
        SSLVerifyDepth 10

        <Directory /var/www/sstt6915_ssl>
            Options Indexes FollowSymLinks MultiViews
            AllowOverride None
            Order allow,deny
            allow from all
        </Directory>
    </VirtualHost>
</IfModule>
```

Cuadro 29: Habilitando ssl

Donde vemos varias diferencias respecto al que proporciona el servicio HTTP:

- Ahora no pedirá usuario y contraseña
- Ahora escuchamos en el puerto 443
- *SSLEngine on* indica que usamos SSL
- Las tres siguientes líneas indican dónde encontrar los archivos necesarios para la certificación del servidor
- Las dos siguientes líneas indican que queremos realizar, también, la autenticación del cliente

De nuevo, debemos habilitar este Virtual Host usando a2ensite y reiniciar el servicio Apache.

Procedemos a hacer la prueba, accediendo desde Firefox a *https://www.sstt6915.org*:

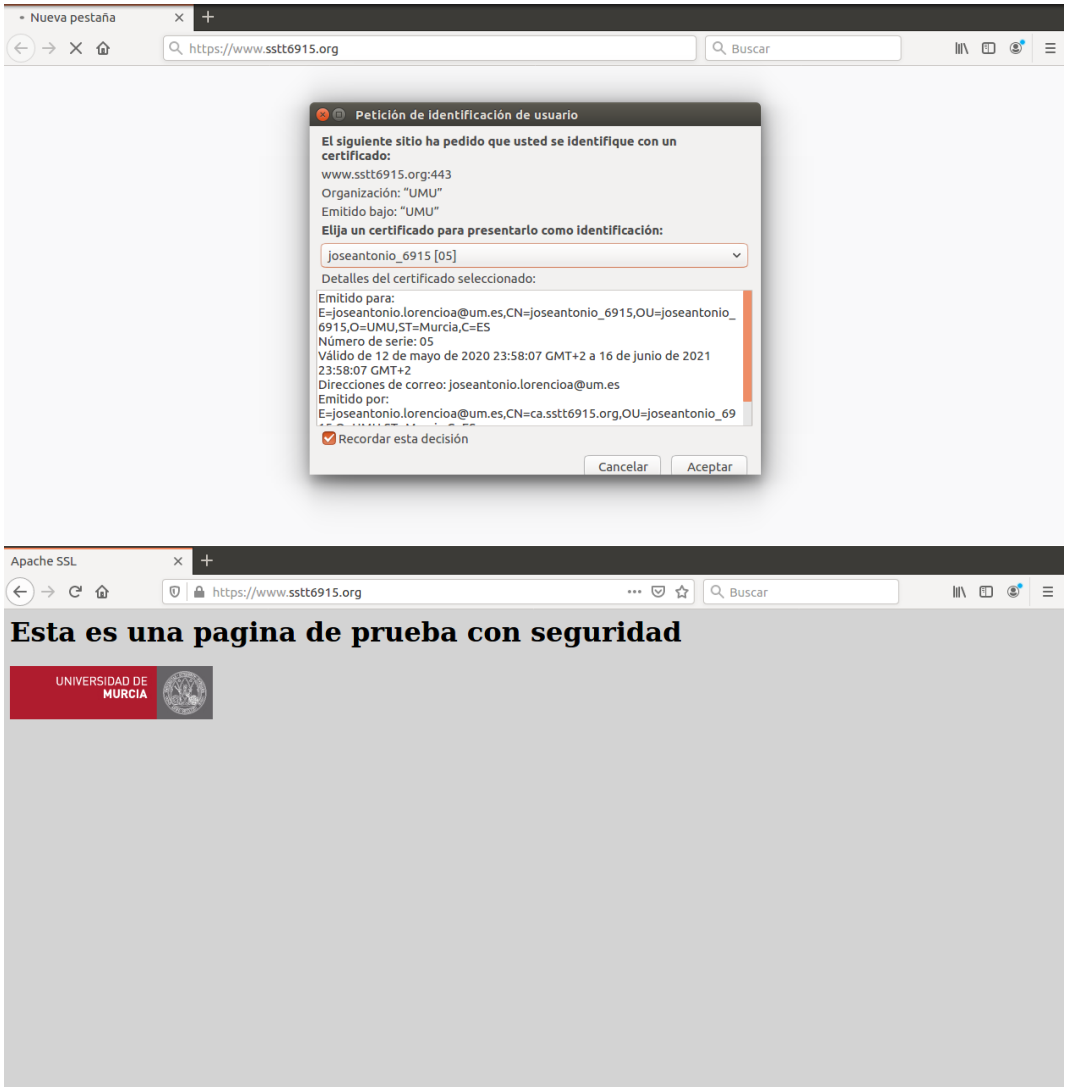


Figura 7: Accediendo al servicio Apache HTTPS

Y comprobamos que el funcionamiento es el deseado.

3.4. IPSEC

Para el despliegue de IPSec hemos usado el Software Strongswan, el cual debemos instalar tanto en la máquina cliente como en el servidor:

```
1        sudo apt-get install strongswan
2
```

Cuadro 30: Instalando StrongSwan

Para su configuración, debemos modificar los ficheros */etc/ipsec.conf* y */etc/ipsec.secrets* también en ambas máquinas.

El primero de los ficheros establece la configuración de la herramienta Strongswan y el segundo contiene la clave compartida.

Vamos a mostrar los ficheros de configuración del servidor, ya que los del cliente son análogos.

```
1         #   ipsec.conf  -   strongSwan IPsec configuration file
2
3         #   basic configuration
4
5         config setup
6             #   strictcrlpolicy=yes
7             #   uniqueids=no
8
9         conn %default
10            ikelifetime=60m #tiempo de vida de una IKE SA
11            keylife=20m #tiempo de vida de una IPsec SA
12            rekeymargin=3m
13            keyingtries=1
14            mobike=no
15            keyexchange=ikev2   #usamos IKEv2
16            authby=pubkey   #autenticación con clave pública
17
18        #para la conexión entre los dos hosts
19        conn host-host
20            left=192.168.56.104
21            leftcert=/etc/ipsec.d/certs/servercert.pem
22            leftid="C=ES, ST=Murcia, O=UMU, OU=joseantonio_6915, CN=www.sstt6915.org, emailAddress=
    joseantonio.lorencioa@um.es"
23            right=192.168.56.101
24            rightid="C=ES, ST=Murcia, O=UMU, OU=joseantonio_6915, CN=joseantonio_6915, emailAddress=
    joseantonio.lorencioa@um.es"
25            type=tunnel
26            auto=start
27            ah=sha1-sha256-modp1024
28        # Add connections here
29
30        # Sample VPN connections
31
```

Cuadro 31: Fichero */etc/ipsec.conf* de la máquina servidor

En el cliente, debemos intercambiar los valores de 'left' por los de 'right'.

Este fichero es bastante autoexplicativo, y algunas líneas tienen un comentario explicando para qué sirven. Solo voy a puntualizar que la línea *type=tunnel* realmente es prescindible, pues este es el modo pro defecto, pero la ponemos para que quede más claro. También mencionar que escribimos *ah* en lugar de *esp*, ya que esta última provee tanto autenticación como encriptación, pero nosotros solo queremos implementar autenticación. Lo que lo acompaña es el protocolo a usar. La última línea indica que IKEv2 se ejecute cuando se haga *ipsec start*.

Vamos a ver el contenido del segundo fichero mencionado:

```
# This file holds shared secrets or RSA private keys for authentication.

# RSA private key for this host, authenticating it to any other host
# which knows the public part.

: RSA /etc/ipsec.d/private/serverkey.pem

```

Cuadro 32: Fichero */etc/ipsec.secrets* de la máquina servidor

La diferencia con el mismo fichero en el cliente es que este contendrá la *clientkey.pem*, en lugar de la *serverkey.pem*. Estas serán las claves que se intercambiarán. Es importante guardar los siguientes archivos en los siguientes directorios:

- La clave privada (*servercert.pem* para el servidor y *clientcert.pem* para el cliente) en */etc/ipsed.d/certs*
- El certificado de cada entidad IKE (*serverkey.pem* para el servidor y *clientkey.pem* para el cliente) en */etc/ipsed.d/private*
-
- El certificado de la CA (*cacert.pem* para el servidor y cliente) en */etc/ipsed.d/cacert*

Una vez hecho todo esto, podemos ejecutar en ambas máquinas *ipsec restart* y las conexiones posteriores ya estarían protegidas. Esto podemos probarlo:

```
root@desktop:/home/alumno# ipsec up host-host
establishing CHILD_SA host-host
generating CREATE_CHILD_SA request 3 [ SA No KE TSi TSr ]
sending packet: from 192.168.56.101[500] to 192.168.56.104[500] (380 bytes)
received packet: from 192.168.56.104[500] to 192.168.56.101[500] (332 bytes)
parsed CREATE_CHILD_SA response 3 [ SA No KE TSi TSr ]
connection 'host-host' established successfully
```

Figura 8: Probando IPsec

Y observamos que funciona correctamente.

4. Descripción de la implementación del servicio web-SSTT HTTP

Para esta explicación, voy a seguir el flujo del programa. Es decir, voy a suponer que un cliente quiere contactar con mi servidor e intercambiar mensajes HTTP, explicando el funcionamiento del servidor en cada caso posible.

El establecimiento de la conexión TCP se realizaba correctamente con el código original proporcionado por el equipo docente. Sabemos que se crea un hijo y este pasa a procesar la petición web. A este procedimiento, *process_web_request*, le he añadido un nuevo parámetro, *int port*, para poder conocer el puerto en el que estoy operando en todo momento.

En este momento nos encontramos dentro de este procedimiento, que procesará peticiones entrantes hasta que:

1. El cliente cierre la conexión TCP
2. Salte el timeout, es decir, el cliente pase más tiempo de lo que dura timeout sin enviar ninguna petición
3. El cliente envíe más cantidad de peticiones de las definidas como máximo (esta es una mejora que he realizado)

Para ello, el procesamiento de peticiones web se encuentra dentro de un bucle de la siguiente forma:

```
void process_web_request(int descriptorFichero, int port)
{
  fd_set client;
  FD_ZERO(&client);
  FD_SET(descriptorFichero, &client);

  struct timeval timeout;
  timeout.tv_sec = TIMEOUT;
  timeout.tv_usec = 0;

  int maxpeticiones=MAXPET;

  while (select(descriptorFichero + 1, &client, NULL, NULL, &timeout))
  {
        //LEER PETICIÓN
        if(LECTURA INCORRECTA){
            exit(-1)
        }
    else
    {
      if(maxpeticiones--<=0){
        //Tratar el caso en que se excede el número máximo de peticiones (MEJORA)
        //ENVIAR code 429 Too Many Requests
        break; //salir del bucle
      }

      //
      //  TRATAR LOS CASOS DE LOS DIFERENTES METODOS QUE SE USAN
      //

      //CHEQUEAR FORMATO DE CABECERA

      if(FORMATO CABECERA INCORRECTO){
          //ENVIAR CODE 400 Bad Request
          continue; //regresar al principio del bucle
      }
      else if (MÉTODO GET) == 0)
      {
        //PROCESAR GET Y RESPONDER
      }
      else if (MÉTODO POST)
      {
        //PROCESAR POST Y RESPONDER
      }
      else{ //Si no es GET ni POST
          //ENVIAR CODE 400 Bad Request
      }

            //Reiniciar timeout
      timeout.tv_sec = 15;
      timeout.tv_usec = 0;
    }
  }
  close(descriptorFichero); //Al acabar el bucle se cierra la conexión TCP
  exit(0);
}
```

Cuadro 33: Esquema general: Persistencia

Aquí muestro un boceto del procedimiento, pero se ve claramente la implementación de la persistencia y de la mejora. Como vemos, creo un conjunto de descriptores de fichero en el que introduzco el descriptor de fichero asociado al socket de conexión con el cliente. Establezco el timeout que considero oportuno (es un #define) y el número máximo de peticiones (otro #define) que puedo atender en una misma conexión TCP. Esta variable se decrementa con cada petición que llega y, al llegar a 0, responderé con un mensaje con código 429 Too Many Requests. Si ocurre esto o salta el timeout, se cerrará la conexión TCP. El cliente deberá establecer una nueva conexión con el servidor para poder continuar enviando peticiones

HTTP. Esto tiene repercusiones en la cabecera de los mensajes HTTP que envío, ahora iremos a ese punto.

Como vemos en el código anterior, el siguiente paso es leer la petición del cliente.

```
    // Definir buffer y variables necesarias para leer las peticiones
    char buffer[BUFSIZE] = {0}; //aquí guardaré el mensaje
    char header[BUFSIZE], body[BUFSIZE]; //aquí guardaré la cabecera y el cuerpo, resp.
    char *reqline[3]; //aquí guardaré los tres primeros campos de la cabecera
    int bytes_leidos; //total bytes leidos
    char *pwd = getenv("PWD"); //Directorio del servidor
    char demanded_url[PATH_MAX], rpath[PATH_MAX]; //url pedida y su resolución en el ordenador

    char host[FILENAME_MAX]; //nombre de host
    sprintf(host, "www.sstt6915.org:%d", port); //Así envío mi nombre de host

    bytes_leidos = leer(buffer, header, body, descriptorFichero); //Leo la petición

    if (bytes_leidos < 0)
    {
      //ERROR DE LECTURA
      exit(-1);
    }
    else if (bytes_leidos == 0)
    {
      //CLIENTE DESCONECTADO
      descriptorFichero);
      exit(-1);
    }
    else
    {
      //LA LECTURA HA IDO BIEN

      if(maxpeticiones--<=0){
        //Tratar el caso en que se excede el número máximo de peticiones (MEJORA)
        strcpy(demanded_url,"/home/alumno/Practicas/errors/429_TooManyRequests.html");
        send_file_from_url(429, "Too Many Requests", demanded_url, host, descriptorFichero);
        //Cierro la conexión por considerarlo un ataque a mi servidor
        break;
      }

      char *start_line = strtok(buffer, "\r\n");
      reqline[0] = strtok(start_line, " \t");
      reqline[1] = strtok(NULL, " \t");
      reqline[2] = strtok(NULL, " \t");

            //COMPROBAMOS EL FORMATO DE LA CABECERA
      if(CABECERA INCORRECTA){
        //ENVIAR ERROR 400 Bad Request
      }
      else if (strncmp(reqline[0], "GET\0", 4) == 0) //Compruebo si es un GET
      {
        //PROCESAR GET
      }
      else if (strncmp(reqline[0], "POST\0", 5) == 0) //Compruebo si es un POST
      {
        //PROCESAR POST
      }
      else{ //Si no es GET ni POST
        //ENVIAR ERROR 400 Bad Request
      }
      //  Como se trata el caso excepcional de la URL que no apunta a ningún fichero
      //  html
      timeout.tv_sec = 15;
      timeout.tv_usec = 0;
    }
  }
  close(descriptorFichero);
  exit(0);
```

Cuadro 34: Esquema general: lectura de la petición

Fijémonos primero en el array *reqline[3]*, en el que guardo los 3 campos de la primera línea del mensaje HTTP. Pongamos ahora nuestro ojo en la función *leer*. Que es así:

```
int leer(char buffer[BUFSIZE], char header[BUFSIZE], char body[BUFSIZE], int descriptorFichero){
  int bytes_leidos, bytes_acum=0; //tamaño leido en cada trama TCP, tamaño total
  int continue_reading=0; //si hay cuerpo, se activa esta variable
  int body_len=0; //si hay cuerpo, aquí guardo su tamaño
  while(((bytes_leidos = read(descriptorFichero, buffer+bytes_acum, BUFSIZE)) > 0) && !strstr(buffer+
    max(bytes_acum-3,0), "\r\n\r\n") && !continue_reading){
    bytes_acum+=bytes_leidos;
    char *con_len=NULL;
    //Si hay cuerpo, no paro en \r\n\r\n, si no cuando he leido todo
    if(!continue_reading && (con_len=strstr(buffer, "Content-Length: "))!=NULL){
      con_len += 16;
      char *endline=strstr(con_len, "\r\n");
      char cl[4];
      strncpy(cl, con_len, endline-con_len);
      continue_reading = body_len = atoi(cl);
    }
    if(continue_reading){
      continue_reading-=bytes_leidos;
    }
  }
  bytes_acum+=bytes_leidos; //Los últimos bytes leidos
  if (bytes_leidos < 0)
  {
    return(-1);
  }
  else if (bytes_leidos == 0)
  {
    return(0);
  }
  else
  {
    //
    // Si la lectura tiene datos válidos terminar el buffer con un \0
    //
    buffer[bytes_acum] = '\0';


    //Si el content_length entró en la última iteración, no lo compruebo dentro del while,
    //tengo que hacerlo fuera
    char *con_len=NULL;
    if(body_len == 0 && (con_len=strstr(buffer, "Content-Length: "))!=NULL){
      con_len += 16;
      char *endline=strstr(con_len, "\r\n");
      char cl[4];
      strncpy(cl, con_len, endline-con_len);
      body_len = atoi(cl);
    }

    char *header_ends = strstr(buffer, "\r\n\r\n");
    int head_len = header_ends-buffer;
    strncpy(header, buffer, head_len);
    if(body_len>0){
      strncpy(body, buffer+head_len+4, body_len);
    }
  }
}
```

Cuadro 35: Procedimiento *Leer* para leer la petición HTTP

Lo que hago es ir leyendo hasta que encuentro "\r\n\r\n", lo que significa que la cabecera la he recibido completamente. Si encuentro el campo *ContentLength* en la cabecera, significa que también hay cuerpo, por lo que debo leer una cantidad adicional de bytes, indicada en ese mismo campo. Para eso sirve la variable *continue_reading*.

Por otro lado, es importante destacar que la última comprobación del while no ejecuta el código interior, por lo que fuera debo sumar los últimos bytes leidos, así como comprobar si hay cuerpo.

Además, hay que notar que he supuesto que el mensaje que me van a enviar ocupa menos de 8K. Esta suposición es razonable en un servidor en pequeña escala como el nuestro, quizá para trabajos mayores habría que deshacerse de esta suposición.

Ahora pasamos a comprobar si la cabecera del mensaje es correcta. Para ello he creado la función *check_header(char header[BUFSIZE], int port)*, que es como sigue:

```
int check_header(char header[BUFSIZE], int port){

      regex_t regex_line;
      char *reqline[3];
      reqline[0]=strtok(header, " \t");
      reqline[1] = strtok(NULL, " \t");
      reqline[2] = strtok(NULL, " \t\n");

      int must=1;
      for(int i=0;i<3;i++){
        if(reqline[i]==NULL || !strcmp(reqline[i]," "))
          must=0;
      }
      if(!must) return 0;

      if(strncmp(reqline[2], "HTTP/1.0", 8) != 0 && strncmp(reqline[2], "HTTP/1.1", 8) != 0){
        return 0;
      }

      char *line = strtok(NULL, "\r\n");

      int match = regcomp(&regex_line, "([a-zA-Z][a-zA-Z0-9]*: ..*)|([a-zA-Z][a-zA-Z0-9]*=..*)",
    REG_EXTENDED);

      int hostexists=0;
      char *is_host=NULL;
      char host[30], ip[30];

      sprintf(host, "www.sstt6915.org:%d", port);
      sprintf(ip, "192.168.56.104:%d", port);

      while(line!=NULL){
        match = regexec(&regex_line, line, (size_t) 0, NULL, 0);

        if (match != 0) {
          regfree(&regex_line);
          return 0;
        }

        if(!hostexists && (((is_host=strstr(line, host))!=NULL) || ((is_host=strstr(line, ip))!=NULL)))
    {
          hostexists=1;
        }
        line = strtok(NULL, "\r\n");
      }
      regfree(&regex_line);
      return hostexists;
}
```

Cuadro 36: Parseo de la cabecera

Como se puede apreciar, he recurrido a expresiones regulares para realizar esta tarea. El funcionamiento es el siguiente:

1. Cojo la primera línea, para lo que utilizo la función strtok. Compruebo que esta línea tiene 3 campos.
2. Compruebo que el tercero de esos campos me indica alguna de las versiones soportadas de HTTP por mi servidor
3. Avanzo de linea
4. Si la linea no es vacía, le paso la expresión regular. Si es vacía, fin, retornar 1.
5. Si hay error, retornar 0. La cabecera es incorrecta
6. Si no hay error, puede ocurrir:
   a) Aun no he encontrado el host. En este caso compruebo si el host se encuentra en esta línea y si contiene un nombre de host correcto, y si es así pongo la variable que indica esto a 1 (true)
   b) Ya he encontrado el host, seguimos
7. Volver al paso 3

Tras comprobar la correción de la cabecera vemos de qué método HTTP se trata. Si es un GET, el código ejecutado será el siguiente:

```
else if (strncmp(reqline[0], "GET\0", 4) == 0) //Compruebo si es un GET
  {
    if (strncmp(reqline[1], "/\0", 2) == 0)
      reqline[1] = "/index.html"; //Si no se indica nada, devuelvo el index.html

    strcpy(demanded_url, pwd);

    strcat(demanded_url, reqline[1]);

    //
    //  Como se trata el caso de acceso ilegal a directorios superiores de la
    //  jerarquia de directorios
    //  del sistema
    //
    realpath(demanded_url, rpath);
    if ((strstr(rpath, pwd) != NULL))
    {
      send_file_from_url(200, "OK", rpath, host, descriptorFichero);
    }
    else
    {
      //Tratar el caso en que acceden a un directorio superior al mio
      strcpy(demanded_url,"/home/alumno/Practicas/errors/403_Forbidden.html");
      send_file_from_url(403, "Forbidden", demanded_url, host, descriptorFichero);
      //Cierro la conexión por considerarlo un ataque a mi servidor
      break;
    }

  }
```

Cuadro 37: Método GET

Recurrimos ahora a *reqline[0]*, si el método es GET la comprobación del código será correcta.

Si en *reqline[1]* hay /, entonces lo cambio por */index.html*. Tomo *demanded_url*, le pongo el pwd y le concateno *reqline[1]*, ya sabemos la ruta relativa del recurso pedido. Con *realpath* obtenemos la ruta absoluta, comprobamos si esta ruta contiene la ruta absoluta del directorio del servidor y, de no ser así, sabemos que están intentando acceder a directorios superiores o paralelos de la jerarquía de directorios, por lo que mandamos un error 403 Forbidden y cerramos la conexión por considerarlo un ataque al servidor.

En otro caso, el recurso pedido estará debajo de nuestro directorio, enviaríamos entonces, en principio (si la extensión es correcta y si existe, lo que se comprueba después), un code 200 OK y el recurso pedido.

Antes de ver cómo funciona el procedimiento *send_file_from_url* voy a explicar el método POST, cuyo código queda así:

```
else if (strncmp(reqline[0], "POST\0", 5) == 0) //Compruebo si es un POST
  {
    char *email=NULL;
    if((email=strstr(body,"email=joseantonio.lorencioa\%40um.es"))!=NULL){
      strcpy(demanded_url, pwd);
      strcat(demanded_url, "/Privado/successful_login.html");
      send_file_from_url(200, "OK", demanded_url, host, descriptorFichero);
    }
    else
    {
      strcpy(demanded_url, pwd);
      strcat(demanded_url, "/login_gone_wrong.html");
      send_file_from_url(200, "OK", demanded_url, host, descriptorFichero);
    }
  }
  else{ //Si no es GET ni POST
    strcpy(demanded_url, "/home/alumno/Practicas/errors/400_BadRequest.html");
    send_file_from_url(400, NULL, demanded_url, host, descriptorFichero);
  }
```

Cuadro 38: POST y método incorrecto

Lo único que tengo que hacer es inspeccionar el cuerpo del mensaje y ver que contiene la línea

email=joseantonio.lorencioa@um.es

Caso de contenerla, envío code 200 OK y la página html diseñada específicamente para este caso, contenida en la carpeta Privado (que realmente no es privada, al menos por el momento).

Si no, envío otra página también diseñada para este caso, que indica que no se ha introducido un email correcto.

También se ve en este snippet de código, qué ocurre cuando la petición no es un GET ni un POST, que no es ni más ni menos que el envío de un error 400 Bad Request.

Vamos a ver ahora la función encargada de enviar las respuestas HTTP, *send_file_from_url*, que la he implementado como sigue:

```
int send_file_from_url(int code, char *codemsg, char *url, char *host, int descriptorFichero)
{
  int fd, bytes_read, size, len;
  char send_buf[BUFSIZE];
  struct stat st;
  char *contype;
  char demanded_url[PATH_MAX];
  strcpy(demanded_url, url);
  strtok(url, ".");
  contype = strtok(NULL, ".");

  //
  //  Evaluar el tipo de fichero que se está solicitando, y actuar en
  //  consecuencia devolviendolo si se soporta u devolviendo el error correspondiente en otro caso
  //  Devuelvo un mensaje con código 400 Bad Request
  //  También puede ocurrir que se requiera un mensaje de este tipo desde otro lugar, por eso el 'or'
  //


  if (((contype = get_extension(contype))==NULL)){
    code = 415;
    codemsg = "Unsupported Media Type";
    contype = "text/html";
    strcpy(demanded_url, "/home/alumno/Practicas/errors/415_UnsMediaType.html");
  }
  else if(code == 400){
    code = 400;
    codemsg = "Bad Request";
    contype = "text/html";
    strcpy(demanded_url, "/home/alumno/Practicas/errors/400_BadRequest.html");
  }
  //
  //  En caso de que el fichero sea soportado, exista, etc. se envia el fichero con la cabecera
  //  correspondiente, y el envio del fichero se hace en blockes de un máximo de  8kB
  //
  if ((fd = open(demanded_url, O_RDONLY)) != -1) //El fichero existe a partir de nuestro directorio
  {
    fstat(fd,&st);
    size = st.st_size;
    // Aquí pedimos que se cree la cabecera correcta
    len = make_header(code, codemsg, url, host, size, contype, send_buf);

    //
    //Asumo que mi cabecera no ocupa 8kB, lo cual tiene sentido, pues mis cabeceras son cortas
    //y caben en send_buf[BUFSIZE]
    //

    write(descriptorFichero, send_buf, len);

    //Aquí vamos enviando el fichero como mucho en bloques de BUFSIZE=8kB
    while ((bytes_read = read(fd, send_buf, BUFSIZE)) > 0)
      write(descriptorFichero, send_buf, bytes_read);
    return 1;
  }
  else
  {
    //Si todo es correcto, pero el fichero no existe a partir del directorio del servidor,
    //enviamos error 404 Not Found
    strcpy(demanded_url,"/home/alumno/Practicas/errors/404_NotFound.html");
    send_file_from_url(404, "Not Found", demanded_url, host, descriptorFichero);
    return 0;
  }
}
```

Cuadro 39: Procedimiento *send_file_from_url*

Los parámetros son el código de envío (200, 400, 403,...), el mensaje asociado a ese código (OK, Bad Request,...), la ruta del fichero a enviar, el nombre del host y el descriptor de fichero asociado al socket en el que vamos a escribir.

Lo primero que hago es ver si el recurso pedido tiene una extensión soportada por nuestro servidor. Para eso creé la función *get_extension*, que explicaré ahora. Si la extensión es incorrecta, cambio el código, la extensión y la ruta del recurso por la de una página html que muestra un error 415. Después de esta comprobación, veo si el código es 400, en tal caso, cambio el mensaje del código, la extensión, y la ruta del recurso por la de una página html que muestra un error 400. Esto lo hago así porque hay varios lugares desde los que me puede llegar que envíe un Bad Request y consideré más sencillo desde el punto de vista lógico hacerlo así. Ahora ya sabemos que tenemos el recurso que queremos enviar a nuestra disposición, ya sea un error 400 o 415, o un archivo a priori correcto. Tratamos de abrirlo para ver si tenemos dicho recurso. Pueden ahora ocurrir dos cosas:

- El fichero se abre correctamente. Usamos la estructura *stat st* para ver el tamaño del recurso a enviar. Ahora creamos la cabecera con la función *make_header*, que también explicaré. Escribimos en el socket la cabecera, y, por último, escribimos en el socket el recurso, en un while porque puede ser mayor que BUFSIZE.
- El fichero no se abre correctamente. Estamos ante un error 404. Cambiamos la ruta del recurso demandado por la de un html que expresa este error y llamamos a esta misma función con los parámetros adecuados para enviar un error 404. Nótese que aquí podríamos entrar en un bucle infinito si este archivo no se encuentra donde indicamos. En mi servidor me aseguro, por supuesto, de que esto no ocurre.

Solo quedan las dos últimas funciones que han quedado en el tintero. La primera de ellas es *get_extension*:

```
1  char* get_extension(char *contype){
2    char *aux="";
3    int i=0;
4    while(strcmp(aux, contype) && i<11){
5      aux = extensions[i].ext;
6      i++;
7    }
8    if(strcmp(aux, contype)){
9      return NULL;
10   }
11   return extensions[i-1].filetype;
12 }
```

Cuadro 40: Función *get_extension*

Es muy sencilla, lo único que hago es pasarle la extensión de un archivo, recorrer la estructura proporcionada en el código original del programa que contiene asociadas extensiones y lo que se envía en el mensaje HTTP para indicar esa extensión. Si encuentro mi extensión en la parte izquierda de la tabla, devuelvo la parte derecha. Si no la encuentro, devuelvo NULL.

Por último, vamos a ver la función *make_header*:

```
int make_header(int code, char *codemsg, char *url, char *host, int conlength, char *contype, char buf
    []){
  int n=sprintf(buf, "HTTP/1.1 %d %s\r\n", code, codemsg);
  n+=sprintf(buf+n, "Connection: Keep-Alive\r\n");

  if(host!=NULL){
    n+=sprintf(buf+n, "Host: %s\r\n", host);
  }

  n+=sprintf(buf+n, "Server: Ubuntu 16.04\r\n");

  if(url!=NULL){
  n+=sprintf(buf+n, "Content-Type: %s\r\n", contype);
  n+=sprintf(buf+n, "Content-Length: %d\r\n", conlength);
  }

  time_t date;
  time(&date);
  struct tm *local=localtime(&date);

  n+=sprintf(buf+n, "Date: %02d/%02d/%d %02d:%02d:%02d\r\n", local->tm_mday, local->tm_mon+1, local->
    tm_year+1900, local->tm_hour, local->tm_min, local->tm_sec);
  n+=sprintf(buf+n, "Keep-Alive: timeout=%d, max=%d\r\n\r\n", TIMEOUT, MAXPET);
  return n;
}
```

Cuadro 41: Función *make_header*

Que es también bastante simple. Le paso el código de envío, su mensaje asociado, la ruta del recurso, el host, el tamaño del cuerpo, la extensión del recurso y el buffer en el que quiero escribir la cabecera. Y simplemente voy escribiendo cada línea, acabándolas con un \r\n y no escribiendo las líneas que no debo. Para la fecha uso la estructura tm y el tipo de datos time_t, que sirven para esto. Escribo el timeout y el número máximo de peticiones que defino al comienzo, y termino con otro \r\n.

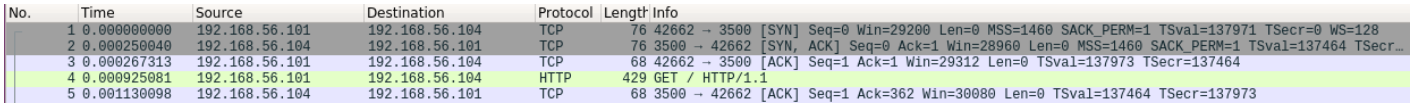
Creo que así queda bastante clara la implementación de mi servidor y todas sus funcionalidades.

5. Trazas representativas de los distintos protocolos empleados

5.1. Trazas del servicio Web-HTTP

Vamos a ver unas cuantas trazas de Wireshark que considero representativas del funcionamiento de mi programa. En el archivo adjunto a la práctica *Captura_EntregaVoluntaria_49196915F.pcap* se puede observar también una traza representativa de lo que sucede. Si ponemos el filtro TCP podemos ver la conexión, así como el correcto funcionamiento de la persistencia. Si ponemos el filtro HTTP podremos observar el intercambio de mensajes HTTP de un par de sesiones TCP distintas.

Pasemos ahora a algunas tramas específicas, en la primera vamos a ver el establecimiento de la conexión TCP, una primera request HTTP GET y la respuesta y todo lo que se desencadena. Por último veremos una request HTTP POST y la respuesta.

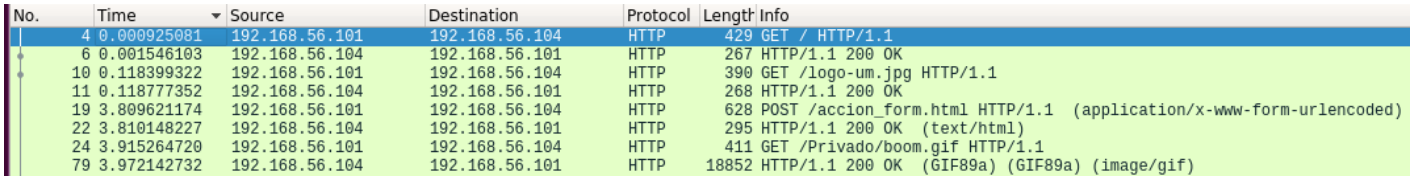


No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.56.101	192.168.56.104	TCP	76	42662 → 3500 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=137971 TSecr=0 WS=128
2	0.000250040	192.168.56.104	192.168.56.101	TCP	76	3500 → 42662 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=137464 TSecr…
3	0.000267313	192.168.56.101	192.168.56.104	TCP	68	42662 → 3500 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=137973 TSecr=137464
4	0.000925081	192.168.56.101	192.168.56.104	HTTP	429	GET / HTTP/1.1
5	0.001130098	192.168.56.104	192.168.56.101	TCP	68	3500 → 42662 [ACK] Seq=1 Ack=362 Win=30080 Len=0 TSval=137464 TSecr=137973

Figura 9: Establecimiento de la conexión y visualización general del intercambio de mensajes

Podemos observar como los dos primeros mensajes son los mensajes TCP SYN de establecimiento de la conexión. Ahora viene un ACK que ya contiene datos HTTP. En este caso una petición GET.

Vamos a pasar a ver los mensajes HTTP que se intercambian.



No.	Time	Source	Destination	Protocol	Length	Info
4	0.000925081	192.168.56.101	192.168.56.104	HTTP	429	GET / HTTP/1.1
6	0.001546103	192.168.56.104	192.168.56.101	HTTP	267	HTTP/1.1 200 OK
10	0.118399322	192.168.56.101	192.168.56.104	HTTP	390	GET /logo-um.jpg HTTP/1.1
11	0.118777352	192.168.56.104	192.168.56.101	HTTP	268	HTTP/1.1 200 OK
19	3.809621174	192.168.56.101	192.168.56.104	HTTP	628	POST /accion_form.html HTTP/1.1 (application/x-www-form-urlencoded)
22	3.810148227	192.168.56.104	192.168.56.101	HTTP	295	HTTP/1.1 200 OK (text/html)
24	3.915264720	192.168.56.101	192.168.56.104	HTTP	411	GET /Privado/boom.gif HTTP/1.1
79	3.972142732	192.168.56.104	192.168.56.101	HTTP	18852	HTTP/1.1 200 OK (GIF89a) (GIF89a) (image/gif)

Figura 10: Mensajes intercambiados para GET y POST

En esta figura observamos la secuencia de mensajes intercambiados. Que es, a alto nivel:

1. GET Index
2. 200 OK Index
3. GET Imagen_Dentro_de_Index
4. 200 OK Imagen_Dentro_de_Index
5. POST Formulario
6. 200 OK Formulario_Aceptado
7. GET Imagen_Dentro_de_Mensaje_Aceptación
8. 200 OK Imagen_Dentro_de_Mensaje_Aceptación

Y podemos ver ahora el interior de alguno de estos mensajes. Vamos a verlo del que nos devuelve un GIF.

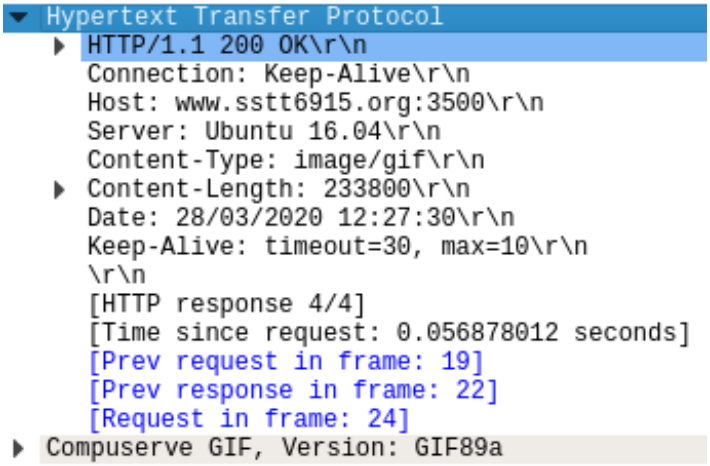


Figura 11: Mensaje HTTP que envía un GIF

Podemos observar los diferentes campos de la cabecera. Como el Connection, o el Keep-Alive, con el que indico el mecanismo de persistencia utilizado. Vemos que en content-type pone gif. Vemos también como en Content-Length se encuentra el tamaño del archivo gif. Y vemos como la cabecera cierra con una línea consistente en un único retorno de carro. Lo último es la forma en que se codifica un GIF, que no tiene interés para lo que estoy explicando.

Ahora voy a forzar un error 404 para poder verlo.

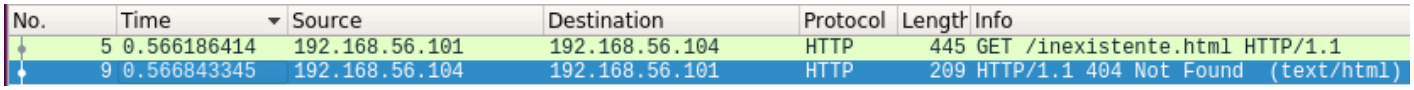


Figura 12: Intercambio de mensajes para forzar un error 404

Y ahora me centro en el propio mensaje de error:

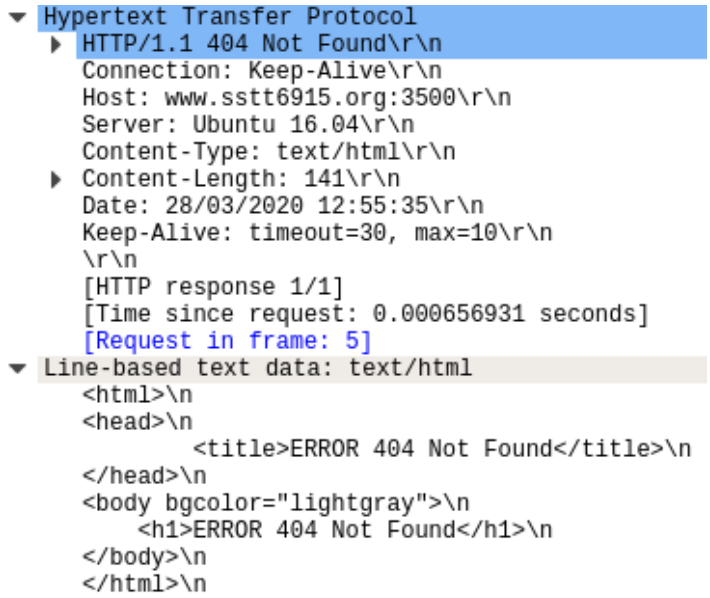


Figura 13: Mensaje de error 404

Seguimos observando la correción de la cabecera. El Content-Length indicando el tamaño del HTML que creé para

este caso, y que puede verse en el último campo del mensaje, el cuerpo.

Para acabar, voy a usar Telnet para pedir un recurso que esté fuera del directorio del servidor. Esto forzará un error 403 Forbidden, así como el cierre de la conexión TCP. Vamos a verlo:

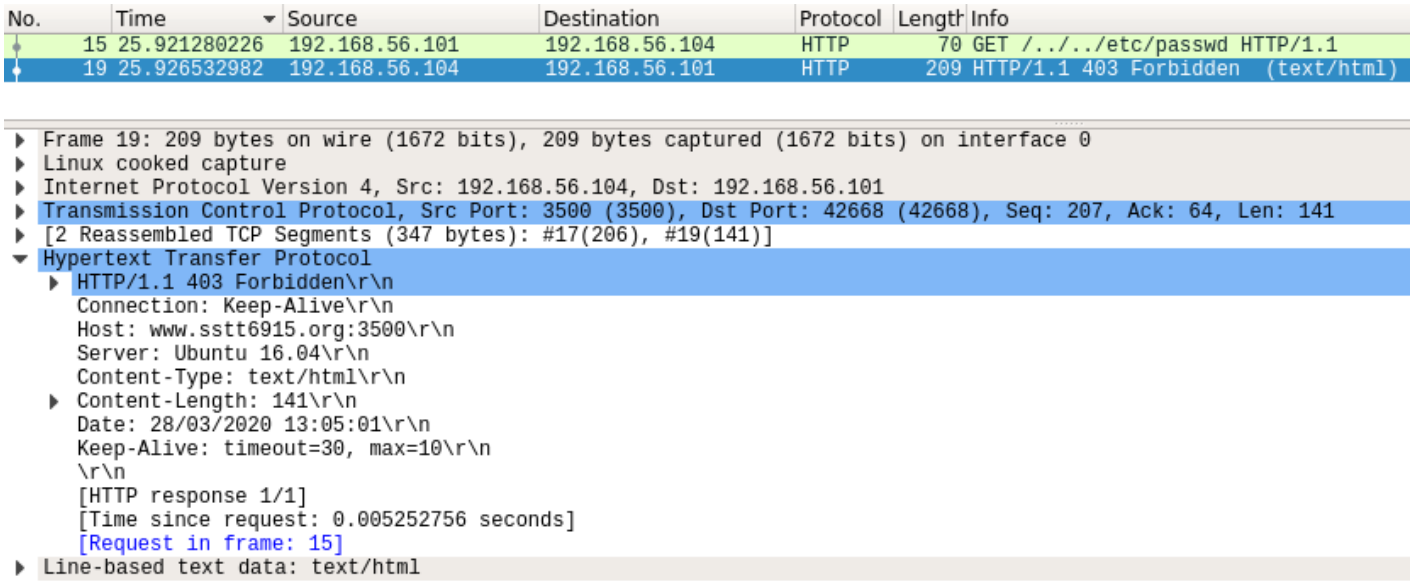


Figura 14: Intercambio HTTP ante una petición de un recurso fuera del servidor

Observamos como efectivamente nos llega el error 303 y podemos ver la cabecera en esta misma figura. Falta ver como se cierra la conexión:

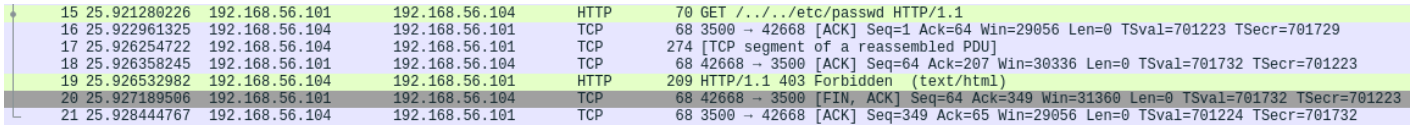


Figura 15: Fin de la conexión tras un error 403 Forbidden

Donde observamos como cuando nos llega el error 403, nos llega también el cierre de la conexión TCP.

Así, hemos visto trazas de momentos representativos por los que pasa nuestro servicio a lo largo de una conexión TCP e intercambio HTTP, desde el establecimiento hasta la finalización, comprobando la corección de las cabeceras y, en general, de los mensajes enviados.

5.2. Trazas de DNS

Ahora vamos a proceder a analizar las trazas de los servicios que no estaban implementados al momento de la entrega anticipada. Para ello, como requiere la práctica, vamos a capturar con wireshark tráfico de todos los servicios implementados. Tras esto, vamos a seleccionar fragmentos representativos de cada servicio y los explicaremos. La traza completa irá adjunta a esta práctica.

En la siguiente captura de Wireshark podemos ver cómo el cliente pide al servidor la dirección IP de *www.sstt6915.org*:

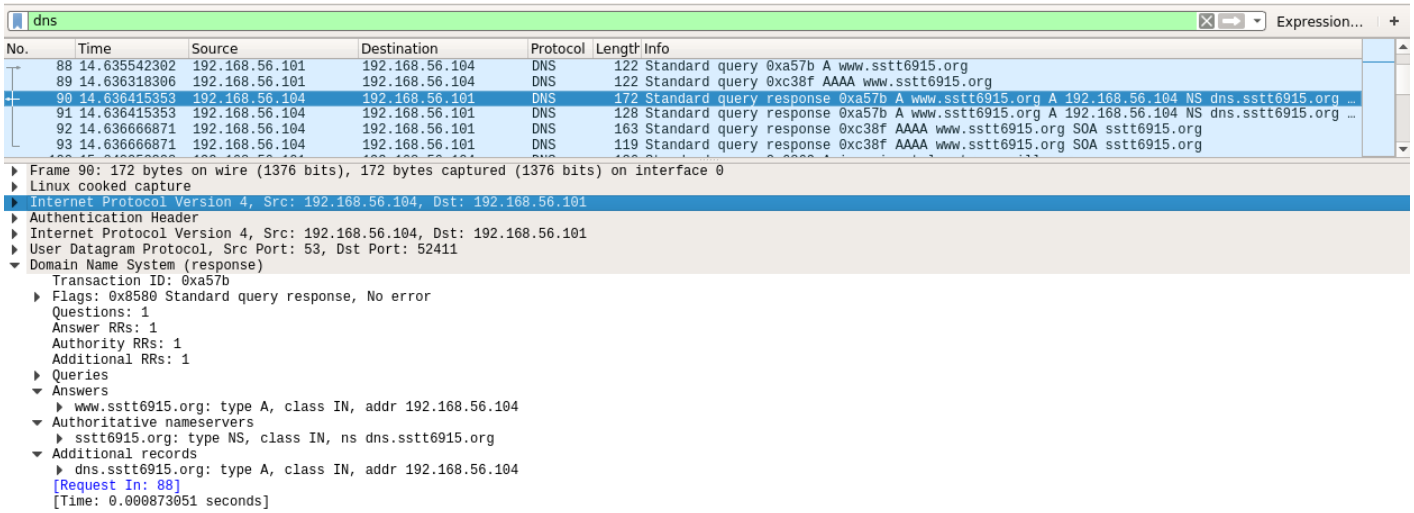


Figura 16: Petición DNS para *www.sstt6915.org*

Observamos como el cliente pide tanto la IPv4 (registro A) como la IPv6 (registro AAAA).

Cuando el servidor recibe esta petición, consulta en su fichero de zona y responde enviando una respuesta indicando el registro A con la dirección pedida. Nótese que a la petición AAAA no responde, pues no dispone de esta información.

Si observamos la traza más adelante, podemos ver una consulta a Facebook, que nos llega correctamente a través del servidor sstt6915.org. Además, esta vez también nos llega la dirección AAAA, pues Facebook sí dispone de esta información para sus servidores.

5.3. Trazas de SMTP/POP

Vamos a analizar un par de fragmentos de la traza:

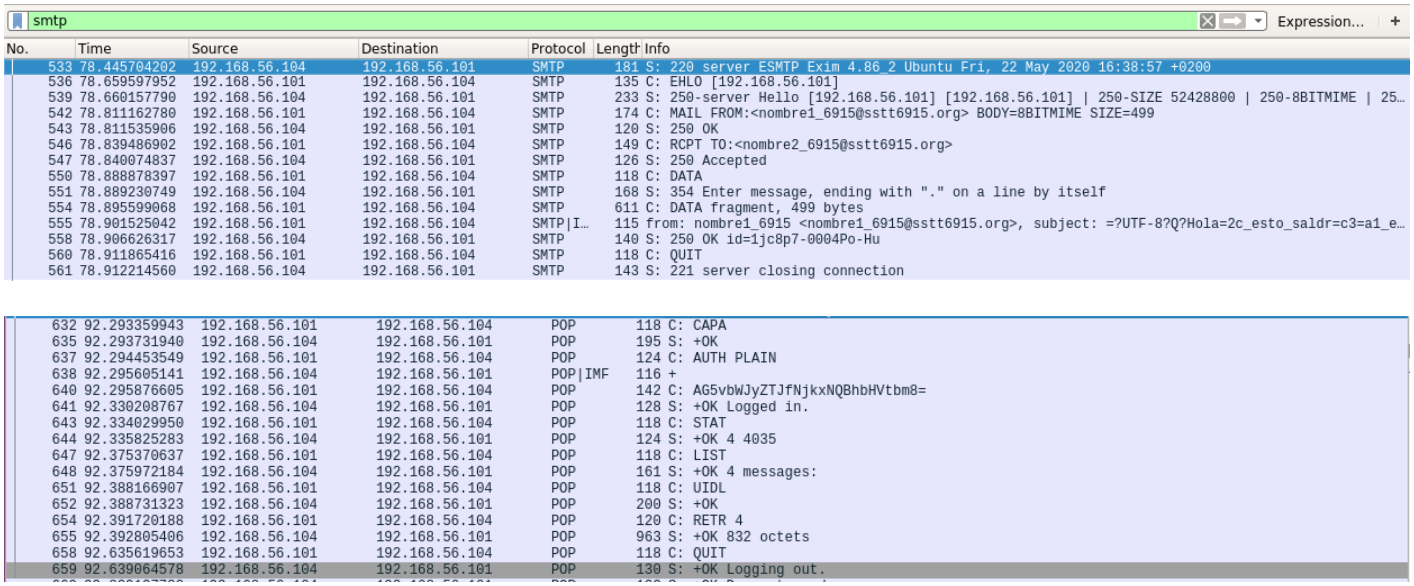


Figura 17: Trazas SMTP/POP

Donde podemos ver que:

1. El cliente comienza la conexión con un EHLO y su IP
2. El servidor contesta con code 250
3. El cliente indica que va a enviar un mensaje desde una dirección de correo, indicando la codificación del cuerpo y su tamaño.
4. El servidor contesta con code 250 OK
5. El cliente indica a qué usuario desea enviar el mensaje
6. El servidor lo acepta, code 250
7. El cliente anuncia que va a enviar los datos del mensaje
8. El servidor responde con code 354, indicando que el mensaje debe terminar con una línea con un único punto "."
9. El cliente envía los datos
10. El servidor anuncia que los ha recibido correctamente
11. El cliente anuncia que ha terminado con el comando QUIT
12. El servidor cierra la conexión

En este momento he cambiado de usuario para recibir los mensajes y verificar que llegaba correctamente mediante POP. Como podemos ver, lo que sucede es lo siguiente:

1. Con CAPA el cliente pide la lista de comandos disponibles
2. El servidor responde con un OK y la lista de comandos disponibles, visibles si abrimos este paquete en la traza
3. El cliente pide autenticación mediante AUTH PLAIN y envía la contraseña del usuario
4. El servidor responde con OK Logged in
5. El cliente hace un STAT, pidiendo información sobre sus mensajes
6. El servidor responde con OK 4 4035, es decir, que hay 4 mensajes y ocupan, en total, 4035 bytes
7. El cliente envía el comando LIST, para que le muestre la lista de los mensajes
8. El servidor responde con dicha lista
9. Con UIDL el cliente pide al servidor la lista de los mensajes, cada uno con su ID
10. El servidor responde de nuevo con esa lista
11. El cliente pide que le dé el mensaje con ID 4
12. El servidor le devuelve este mensaje
13. Una vez tiene el mensaje que quería, hace QUIT, para cerrar la conexión
14. El servidor responde con un OK y cierra la conexión

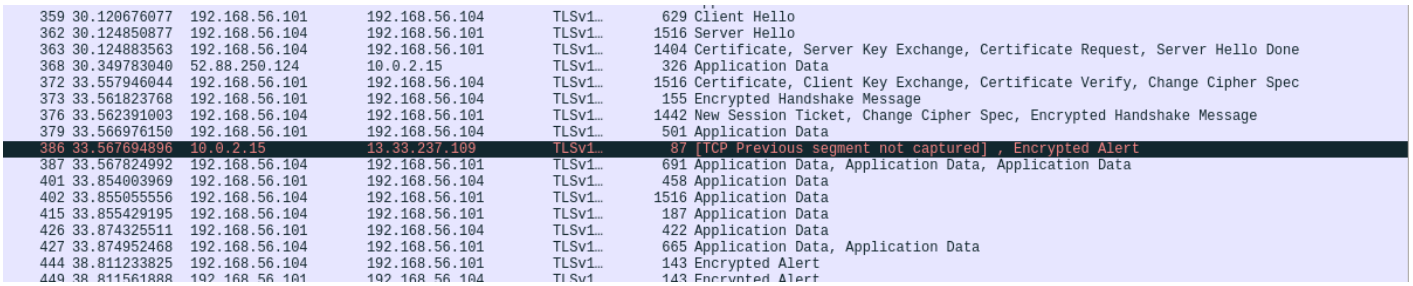
5.4. Trazas de HTTP/HTTPS

Sobre HTTP no hay mucho que comentar. El intercambio de mensajes es el rutinario. La única observación es que si abrimos el mensaje GET, veremos que introduce el usuario y la contraseña que se requieren para la autenticación.

117	16.855473506	192.168.56.104	192.168.56.101	HTTP	854	HTTP/1.1 401 Unauthorized (text/html)
126	18.943559916	192.168.56.101	192.168.56.104	HTTP	511	GET / HTTP/1.1
127	18.944639781	192.168.56.104	192.168.56.101	HTTP	622	HTTP/1.1 200 OK (text/html)
130	19.488513029	192.168.56.101	192.168.56.104	HTTP	467	GET /logo-um.jpg HTTP/1.1
138	19.571520048	192.168.56.104	192.168.56.101	HTTP	129	HTTP/1.1 200 OK (JPEG JFIF image)
155	19.893010721	192.168.56.101	192.168.56.104	HTTP	432	GET /favicon.ico HTTP/1.1
156	19.893670968	192.168.56.104	192.168.56.101	HTTP	606	HTTP/1.1 404 Not Found (text/html)
159	22.530129532	192.168.56.101	192.168.56.104	HTTP	554	GET /ref.html HTTP/1.1
160	22.530980491	192.168.56.104	192.168.56.101	HTTP	591	HTTP/1.1 200 OK (text/html)

Figura 18: Traza HTTP

Respecto a la traza HTTPS:



359	30.120676077	192.168.56.101	192.168.56.104	TLSv1…	629	Client Hello
362	30.124850877	192.168.56.104	192.168.56.101	TLSv1…	1516	Server Hello
363	30.124883563	192.168.56.104	192.168.56.101	TLSv1…	1404	Certificate, Server Key Exchange, Certificate Request, Server Hello Done
368	30.349783040	52.88.250.124	10.0.2.15	TLSv1…	326	Application Data
372	33.557946044	192.168.56.101	192.168.56.104	TLSv1…	1516	Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec
373	33.561823768	192.168.56.101	192.168.56.104	TLSv1…	155	Encrypted Handshake Message
376	33.562391003	192.168.56.104	192.168.56.101	TLSv1…	1442	New Session Ticket, Change Cipher Spec, Encrypted Handshake Message
379	33.566976150	192.168.56.101	192.168.56.104	TLSv1…	501	Application Data
386	33.567694896	10.0.2.15	13.33.237.109	TLSv1…	87	[TCP Previous segment not captured] , Encrypted Alert
387	33.567824992	192.168.56.104	192.168.56.101	TLSv1…	691	Application Data, Application Data, Application Data
401	33.854003969	192.168.56.101	192.168.56.104	TLSv1…	458	Application Data
402	33.855055556	192.168.56.104	192.168.56.101	TLSv1…	1516	Application Data
415	33.855429195	192.168.56.104	192.168.56.101	TLSv1…	187	Application Data
426	33.874325511	192.168.56.101	192.168.56.104	TLSv1…	422	Application Data
427	33.874952468	192.168.56.104	192.168.56.101	TLSv1…	665	Application Data, Application Data
444	38.811233825	192.168.56.104	192.168.56.101	TLSv1…	143	Encrypted Alert
449	38.811561888	192.168.56.101	192.168.56.104	TLSv1	143	Encrypted Alert

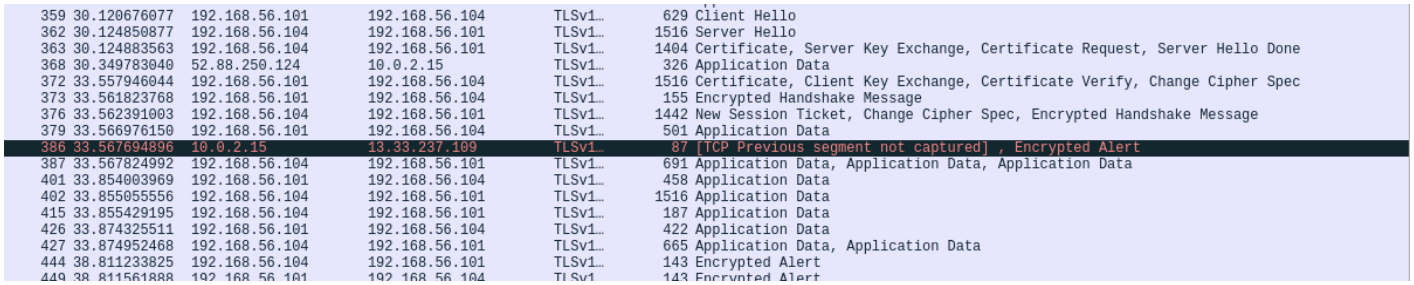
Figura 19: Traza HTTPS

Esta si tiene algunos aspectos que podemos comentar:

1. El cliente envía un mensaje ClientHello, que contiene información criptográfica para hacer efectiva la autenticación, indicando las suites de cifrado que soporta
2. De entre las opciones que indica el cliente, el servidor elige los parámetros de la conexión y los indica en su mensaje ServerHello
3. Se produce el intercambio de los certificados x509
4. Una vez verificados los certificados, se negocia la clave usada para cifrar los datos intercambiados, lo que se conoce como *master secret*
5. Una vez hecho esto, se intercambian los datos de la aplicación subyacente, en nuestro caso son páginas web. Por supuesto, estos mensajes están cifrados.

5.5. Trazas de IPSec

Por último, vamos a ver una traza en la que podamos ver el funcionamiento de IPSec.



359	30.120676077	192.168.56.101	192.168.56.104	TLSv1…	629	Client Hello
362	30.124850877	192.168.56.104	192.168.56.101	TLSv1…	1516	Server Hello
363	30.124883563	192.168.56.104	192.168.56.101	TLSv1…	1404	Certificate, Server Key Exchange, Certificate Request, Server Hello Done
368	30.349783040	52.88.250.124	10.0.2.15	TLSv1…	326	Application Data
372	33.557946044	192.168.56.101	192.168.56.104	TLSv1…	1516	Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec
373	33.561823768	192.168.56.101	192.168.56.104	TLSv1…	155	Encrypted Handshake Message
376	33.562391003	192.168.56.104	192.168.56.101	TLSv1…	1442	New Session Ticket, Change Cipher Spec, Encrypted Handshake Message
379	33.566976150	192.168.56.101	192.168.56.104	TLSv1…	501	Application Data
386	33.567694896	10.0.2.15	13.33.237.109	TLSv1…	87	[TCP Previous segment not captured] , Encrypted Alert
387	33.567824992	192.168.56.104	192.168.56.101	TLSv1…	691	Application Data, Application Data, Application Data
401	33.854003969	192.168.56.101	192.168.56.104	TLSv1…	458	Application Data
402	33.855055556	192.168.56.104	192.168.56.101	TLSv1…	1516	Application Data
415	33.855429195	192.168.56.104	192.168.56.101	TLSv1…	187	Application Data
426	33.874325511	192.168.56.101	192.168.56.104	TLSv1…	422	Application Data
427	33.874952468	192.168.56.104	192.168.56.101	TLSv1…	665	Application Data, Application Data
444	38.811233825	192.168.56.104	192.168.56.101	TLSv1…	143	Encrypted Alert
449	38.811561888	192.168.56.101	192.168.56.104	TLSv1…	143	Encrypted Alert

Figura 20: Traza IPSec

Vemos como el proceso consiste en dos fases, la fase *IKE_SA_INIT* y la *IKE_AUTH*. Tras esto, la conexión irá en modo túnel verificada por IPSec.

Para comprobar que la conexión, efectivamente, funciona como queremos (es decir, con seguridad pero sin codificar), podemos tomar cualquier paquete que se envíen, por ejemplo los POP, y ver que tiene el Authentication Header, pero el contenido del mensaje es legible, no está cifrado.

6. Problemas encontrados en el proceso de desarrollo del escenario

Los problemas que he tenido fueron todos referidos a la primera parte de la práctica (la parte de la entrega opcional). Ya que la segunda parte, yendo con cuidado y siguiendo los pasos de las prácticas, es muy sencilla. Vuelvo a exponer aquí los problemas que tuve para dejar la práctica compactificada en un único documento.

- **Comprender el trabajo del servidor:** al principio me encontraba un poco perdido, pues sentía que programaba sin un objetivo final claro. Hasta que no me esforcé en entender bien el protocolo HTTP me costó ver cómo debía proceder de forma global.
- **Manejo de buffers, lectura y escritura de mensajes:** esta ha sido la parte que más quebraderos de cabeza me ha producido. Me he apoyado en diversas librerias de C, como *string.h* o *regex.h*, pero aún así he tenido dificultades. Hay muchas comprobaciones a tener en cuenta a la hora de procesar una petición, y otras tantas en las que hay que tener cuidado al generar una respuesta. En ocasiones, sentía que solo estaba poniendo un parche encima de otro para ir arreglando problemas de funcionamiento, finalmente esto me obligó a redefinir desde cero algunas funciones y a crear algunas otras. Al menos me he familiarizado bastante bien con el manejo de strings y buffers en C en todo este proceso.
- **La persistencia:** no me ha producido excesivos problemas, aunque cuando la implementé, al principio, no funcionaba. El fallo no estaba en la implementación de esta, sino en que aún no había programado correctamente el envío de las cabeceras de respuesta. Una vez hice esto, funcionaba correctamente.
- **Uso de variables innecesarias:** a lo largo del desarrollo, he usado distintas variables que consideraba útiles en un principio, pero al avanzar en el proyecto dejaban de serlo y pasaban a ser ineficiencias del programa. Supongo que esto se debe a que debo planificar más a largo plazo mi programa en lugar de 'ir haciendo lo que toca'.

7. Tiempo de trabajo

Tiempo dedicado a la Entrega Anticipada		Tiempo dedicado a la segunda parte de la práctica	
Programación y testeo	Memoria	Configuración y testeo	Memoria
14 horas de trabajo autónomo 6 horas de trabajo en clase	4 horas de trabajo autónomo	12 horas de trabajo autónomo	7 horas de trabajo autónomo

Cuadro 42: Desglose del tiempo de trabajo

8. Conclusión y valoración personal

Tras acabar esta práctica, siento un sabor agridulce. La primera parte de la práctica fue bastante estimulante y realmente considero que son este tipo de prácticas las que deben encargarse en Ingeniería Informática. Son prácticas que te obligan a pensar y a buscar soluciones, a debatir con los compañeros y a lidiar con ls frustración de que lo que has pensado no sea correcto.

La parte agridulce es la segunda. Si bien es cierto que es importante que nos familiaricemos con todas estas tecnologías que hemos estudiado, no estoy muy seguro de que la fórmula para esto sea seguir un tutorial y ya está. Quizás sería más conveniente estudiar solo un par de tecnologías, pero entenderlas en profundidad. Trabajarlas sin seguir tutoriales y haciéndonos pensar un poco.

Evidentemente, el factor de ocurrir una pandemia mundial que nos ha impedido dar clase con normalidad ha influido también en el desarrollo de las prácticas, pero esto no debe ser excusa para intentar mejorar las cosas de cara a los siguietes cursos y seguir hacia delante.