UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Práctica/Trabajo 3: VPN con "simpletun"

PRÁCTICA RCO

Grado en Ingeniería Informática

Autor: Julio Pons Terol
Grupo: 1

Curso 2024-2025

Resumen

En este documento se presenta la práctica 3,

Palabras clave: Simpletun, ...

Abstract

This document presents practice 3

Key words: Simpletun, ...

Índice general

Índice de figuras

Índice de Listados

CAPÍTULO 1
Introducción

Los interfaces Tun/Tap son una característica ofrecida originalmente por Linux (disponible también en otras plataformas como Windows, MacOSX, iOS, Android...) que permite hacer "userspace networking", esto es, permite que programas usuario puedan manipular el tráfico de red a nivel de Ethernet (tap) o a nivel de IP (tun).

Esta práctica explica cómo funcionan los interfaces tun/tap utilizando un programa sencillo de ejemplo: simpletun. Este documento está basado en "Tun/Tap interface tutorial" (`http://backreference.org/2010/03/26/tuntap-interface-tutorial/`)

Simpletun es un programa que puede actuar tanto como cliente como servidor y crea un túnel TCP entre ambos.

1.1 Objetivos

El objetivo principal de este documento es presentar la práctica/trabajo 3.

CAPÍTULO 2
Configuración

Vamos a mantener el esquema de red de la práctica anterior (figura 2.1) pero desactivando el PPTP, las rutas IP y las redirecciones de puertos que se realizaron. Configuraremos RCO-noX como un servidor simpletun.

El campo "nº grupo" debe ser sustituido por el número asignado al grupo de prácticas, por ejemplo, si el número de grupo fuese el 1 los rangos de IPs de las redes VNnet1 y VMnet2 serían 10.24.1.0/24 y 10.54.1.0/24.

Para poder acceder a un servidor que está en una red local (con NAT), es necesario configurar una redirección de puertos en su router para que sea accesible desde la WAN.

Podemos apreciar en la figura 2.1 que el puerto que redireccionamos es el puerto TCP 55555, que es el que utiliza el programa simpletun.

RCO-X actuará como cliente y se conectará al servidor usando la IP de la WAN del router ddwrt-noX.

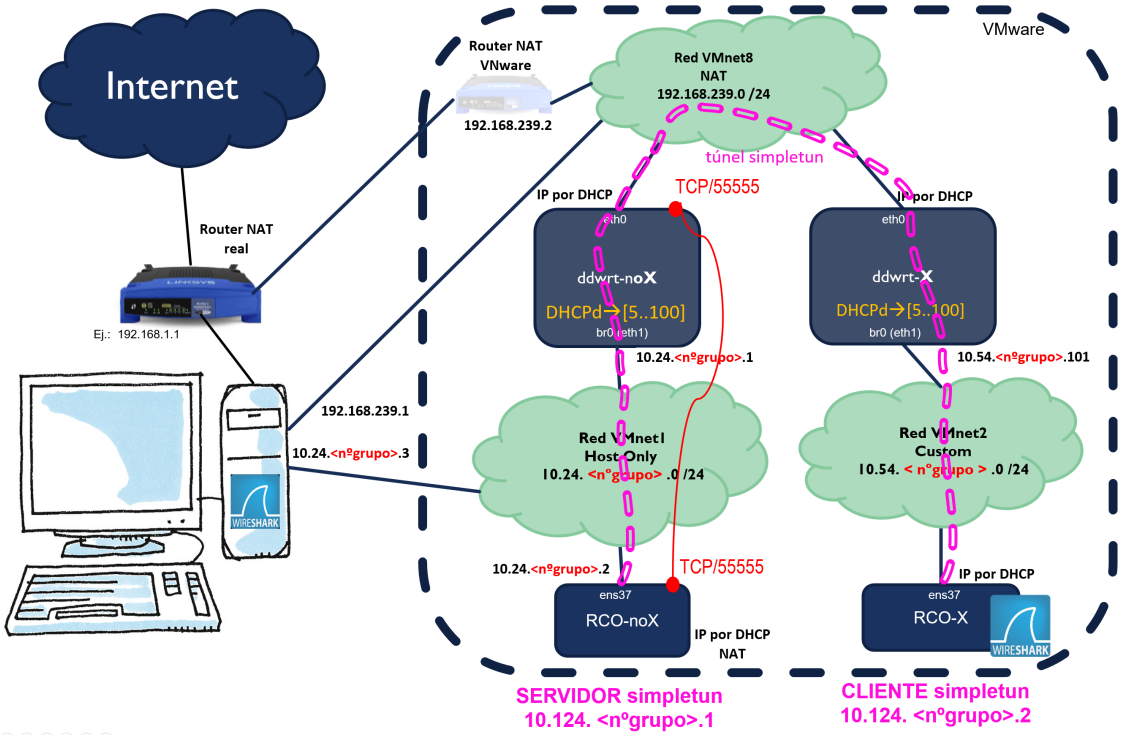

Figura 2.1: Esquema de las máquinas virtuales para conexión simpletun

2.1 Configuración de la redirección de puertos

Todas las configuraciones deben realizarse con las 4 máquinas virtuales encendidas. En la interfaz web de ddwrt-noX indicamos que el tráfico que entre por el puerto TCP 55555 se dirija a este mismo puerto de RCO-noX (dirección IP 10.24.nºgrupo.2), como ejemplo usaremos el nºgrupo=1. Ver figura 2.2.

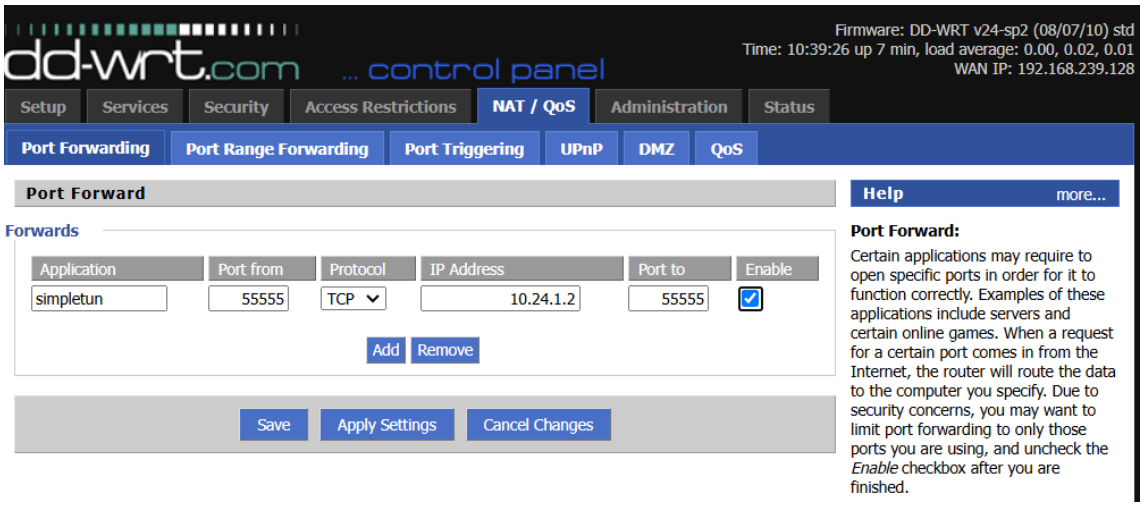

Figura 2.2: Creación de la redirección de puertos

Se puede verificar que la redirección se ha creado correctamente desde la terminal (desde mobaxterm) del ddwrt-noX con la orden iptables, tal y como se muestra en la figura 2.3.

```
root@DD-WRT:~# iptables -nL -t nat
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0            192.168.239.128     tcp dpt:8080 to:10.24.1.1:80
DNAT       tcp  --  0.0.0.0/0            192.168.239.128     tcp dpt:22 to:10.24.1.1:22
DNAT       icmp --  0.0.0.0/0            192.168.239.128     to:10.24.1.1
DNAT       tcp  --  0.0.0.0/0            192.168.239.128     tcp dpt:55555 to:10.24.1.2:55555
TRIGGER    0    --  0.0.0.0/0            192.168.239.128     TRIGGER type:dnat match:0 relate:0
```

Figura 2.3: Orden iptables para ver la tabla nat

2.2 Preparación de CentOS para trabajar con tun/tap

Esta tarea debe hacerse en las dos máquinas RCO. Instalamos gcc, el compilador de c y c++

```
# yum install gcc
```

Descargamos y compilamos simpletun.c.

```
# cd
# mkdir simpletun
# cd simpletun/
# wget https://redescorporativas.es/simpletun.c
# make simpletun
```

También tenemos una copia de simpletun.c en la web oficial `https://web.ecs.syr.edu/~wedu/seed/Labs/VPN/files/simpletun.c`

2.3 Cómo funciona Tun/Tap

A diferencia de otros interfaces de red, como el ens37 que gestionan el hardware Ethernet, o Wlan que gestionan el hardware WiFi, los interfaces Tun y Tap no gestionan ningún harware, son interfaces a dispositivos software. Por lo tanto, no hay ningún "cable" conectado a ellos.

Podemos pensar en los interfaces Tun y Tap como si fueran interfaces normales que cuando el S.O. decide que es el momento de mandar datos "por su cable", lo que realmente hace es mandar los datos a un "programa del usuario" que está vinculado a la interface (mediante el procedimiento que describimos en la práctica). Esto permite al "programa del usuario" (en nuestro caso será el simpletun) procesar los datos y mandarlos por una interface real (por ejemplo, ens37).

La diferencia entre una interface tap y una interface tun es que el interface tap maneja tramas Ethernet, mientras que el tun maneja datagramas IP. Tap es adecuado para hacer puentes y tun para hacer túneles.

2.4 Guía de configuración de las interfaces y activación del túnel simpletun

NOTA IMPORTANTE: Todos los pasos que se indican a continuación tienen naturaleza temporal y desaparecen cuando se reinicia el equipo. Si queremos poder repetirlos deberíamos crear un script que los lanzara todos a la vez.

Asumimos que estamos en el grupo 1.

Vamos a montar un túnel típico como el de la figura 2.4.

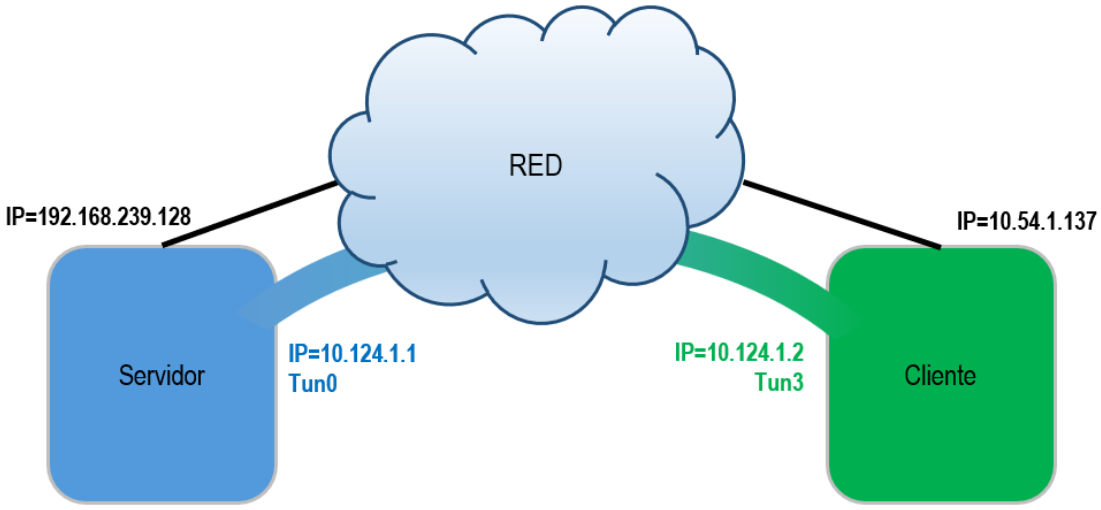

Figura 2.4: Esquema ejemplo de túnel simpletun

El túnel utiliza la red 10.124.1.0/30. que realmente no existe, sólo a efectos de redireccionamiento.

A continuación se muestran los pasos a realizar en ambas máquinas. Observe que una máquina ejecuta el programa de usuario simpletun en modo servidor y la otra en modo cliente

2.4.1. Servidor

IP pública 192.168.239.128 (IP de ddwrt-noX), IP privada 10.24.1.2 (usada de forma indirecta gracias a la redirección de puertos).

1. Creamos tun0 y lo configuramos para que esté asociado a la IP 10.124.1.1

```
# ip tuntap add dev tun0 mode tun
# ip link set tun0 up
# ip addr add 10.124.1.1/30 dev tun0
```

2. Lanzamos el servidor

```
./simpletun -i tun0 -s &
```

   El servidor queda a la espera de que se conecte un cliente a la IP pública en el puerto por defecto 55555.

2.4.2. Cliente

IP 10.54.1.137 (IP de RCO-X irrelevante):

1. Creamos tun3 y lo configuramos para que esté asociado a la IP local 10.124.1.2

```
# ip tuntap add dev tun3 mode tun
# ip link set tun3 up
# ip addr add 10.124.1.2/30 dev tun3
```

2. Lanzamos el cliente

```
./simpletun -i tun3 -c 192.168.239.128 &
```

   El cliente conecta con el servidor en la IP 192.168.239.128 y entra en bucle de escucha de la conexión y del interfaz tun3.

Una vez tengamos el túnel operativo, vamos a añadir reglas de routing para que podamos hacer que las máquinas comuniquen con sus IPs privadas originales.

En este caso crearemos las reglas con la orden ip.

En RCO-noX:

```
# ip route add 10.54.1.0/24 dev tun0
```

En RCO-X

```
# ip route add 10.24.1.0/24 dev tun3
```

En realidad, para este primer objetivo, la máscara debería ser IP/32, poniendo como IP la del otro host pues vamos a comunicar solo con un destinatario, pero el siguiente objetivo será convertir el túnel en site-to-site usando los RCOs como routers.

CAPÍTULO 3
Tareas

3.1 Tarea-1: Comprobación del túnel

Capture tráfico que muestre el encapsulado túnel y explique el funcionamiento de simpletun en base a lo capturado.

Nota: El túnel que hace simpletun no es estándar, por lo que wireshark no podrá decodificar el datagrama contenido en el segmento TCP. Sugerimos que:

1. Con el wireshark en RCO-X capture tráfico de los adaptadores ens37 y tun3 (ordene por tiempo).

2. Realice desde RCO-X un ping –s 1000 10.124.<nºRouter>.1 de tal manera que le sea fácil identificar los paquetes ICMP por el tamaño.

   En el wireshark debería ver por línea de salida del ping, y en este orden (ordenando por tiempo) 1 ICMP echo request, 8 TCPs y 1 ICMP echo reply. Explique porqué. Explique también por qué los ICMP echo no llevan cabecera ethernet.

   Identifique la función de cada uno de los segmentos TCP y verifique (mirando los valores en hexadecimal) que el campo de datos del segmento TCP coincide con el datragrama IP del ICMP echo correspondiente.

3. Para verificar que la regla de routing añadida funciona correctamente, realice desde RCO-X un ping –s 1000 10.24.<nºRouter>.2. Observe con el wireshark las direcciones IP fuente y destino. Justifique el porqué se utilizan esas direcciones.

4. Vamos a cambiar la reglas reglas de routing por las siguentes (adaptelas a su número de grupo y a la IP de RCO-X)

   En RCO-noX:

```
# ip route del 10.54.1.0/24 dev tun0
# ip route add 10.54.1.0/24 dev tun0 src 10.24.1.2
```

   En RCO-X

```
# ip route del 10.24.1.0/24 dev tun3
# ip route add 10.24.1.0/24 dev tun3 src 10.54.1.137
```

5. Realice desde RCO-X un ping –s 1000 10.24.<nºRouter>.2. Observe con el wireshark las direcciones IP fuente y destino. Justifique el porqué se utilizan esas direcciones

3.2 Tarea-2: Usar el túnel como site to site

Vamos a hacer que RCO y RCO-X funcionen como si fuesen routers interconectados por un túnel simpletun.

Los pasos a seguir son:

1. Active el forwarding en ambos RCOs. Los sistemas Linux tiene opción de activar el forwarding, que implica que cuando reciben un datagrama IP cuya IP de destino no es ninguna de las suyas, en vez de descartar el datagrama, lo reenvían aplicando las reglas de routing. En Almalinux esto se puede hacer con la siguiente línea de órdenes:

```
# echo 1 | cat >/proc/sys/net/ipv4/ip_forward
```

   Cambiando el 1 por un 0 se puede desactivar.

   Es recomendable averiguar (y documentar) la forma de hacer esta cambio permanente.

2. Cree reglas en el PC y en el router ddwrt-noX para que el tráfico dirigido a la red 10.54.<nºgrupo>.0/24 use como Gateway 10.24.<nºgrupo>.2

3. Cree reglas en ddwrt-X para que el tráfico dirigido a 10.24.<nºgrupo>.0/24 use como Gateway 10.54.<nºgrupo>.x (la IP de ens37 de rco-X).

   Verifique con órdenes ping que todos los 10.24 comunican con todos los 10.54. Realice alguna captura con wireshark en RCO-X para comprobar el funcionamiento, por ejemplo un ping desde ddwrt-X a ddwrt-noX (usando las IPs de sus LANs).

   En el wireshark debería ver por línea de salida del ping, y en este orden (ordenando por tiempo) 2 ICMP echo request, 8 TCPs y 2 ICMP echo reply. Explique por qué. Explique también porqué uno de los ICMP echo lleva cabecera ethernet y el otro no.

3.3 Tarea-3: Entender cómo se puede programar TunTap

Modificar el código simpletun.c para incluir un cifrado simple del túnel (con esto tendremos "casi" una VPN).

El cifrado más sencillo es un Caesar. Recordemos que Caesar lo que hace es desplazar el código. Por ejemplo, un Caesar-1 codifica la "A" como "B", la "B" como "C", etc. Implemente un Caesar-N, donde N es el número del grupo asignado en prácticas anteriores.

Si a y c son variables de tipo char, las expresiones en C serían:

Codificación Caesar-N: c = (a + N) % 256

Decodificación Caesar-N: a = (c + 256 – N) % 256

El cifrado se debe hacer cuando la información venga del túnel, antes de mandarla por el socket, y viceversa para el descifrado.

Ojo, aunque a modo de ejemplo hablamos de caracteres, los datos que se envían y reciben son binarios por lo que el cifrado se aplica a todos los bytes.

Incluya en el trabajo un listado de la parte de código modificada que incluya comentarios.

Pruebe a mandar un ping por el túnel. Debe funcionar perfectamente. Capture un ping cifrado. Repita el mismo procedimiento (1)(2)(3) de la tarea-1. Verifique que el contenido del segmento TCP ahora coincide con el del ping desplazado en +N caracteres.

3.4 Tarea-4: Cifrado "secreto".

Además del Caesar, debe proponer otro método de cifrado "algo más elaborado". Puede ser (p.ej.) un XOR de los datos con un fichero del sistema operativo (disponible en el equipo emisor y en el receptor), puede ser utilizar módulos de RC4, o de cualquier otro cifrador estándar (disponibles en la red, p.ej en `https://github.com/`). Se valorará la dificultad y "potencia" del método implementado.

Al igual que en el apartado anterior Incluya en el trabajo un listado de la parte de código modificada que incluya comentarios.

Pruebe a mandar un ping o un fichero pequeño de texto por el túnel. Debe de funcionar perfectamente.

Capture los datos cifrados y descifrados dando prueba del buen funcionamiento.

CAPÍTULO 4
Documentación

El trabajo se redactará en la plataforma online overleaf.com cargando una plantilla que se facilitará en la sección de recursos del poliformat (Trabajo3.zip).

El documento debe redactarse orientado a un público con conocimientos técnicos, pero que no tiene porqué conocer el túnel simpletun. No debe ir orientado al profesor.

La plantilla que se os facilita incluye los siguientes apartados, que son los que deberá tener vuestro trabajo:

- Portada con el título "VPN con simpletun", los nombres de los componentes del grupo, el número del grupo y el curso académico.
- Resumen

  Debe ser un breve resumen de todo el trabajo realizado, por lógica es la última parte que se rellena en el documento, aunque se presente al principio. Debe servir para que el lector decida si le interesa continuar leyendo.

  El resumen se completa con las palabras clave, que se utilizan para indexar el documento en una base de datos bibliográfica, hay que utilizar palabras específicas que se relacionen con el trabajo.

  El Abstract y las keywords son la traducción al inglés del resumen y las palabras clave.
- Tabla de contenidos. Es el índice del documento se genera automáticamente. Incluirá el índice de figuras y el índice de listados.
- Introducción

  Debe poner al lector en el contexto del trabajo. Explica los objetivos y qué es lo que se va a hacer en el trabajo y debe incluir el esquema de la red virtual con TODAS las direcciones IP utilizadas, personalizadas según lo indicado.

  A diferencia del resumen, aquí se indica que es lo que se va a hacer en las siguientes secciones, no pretende ser un resumen del trabajo, es más una planificación.

  En la sección de recursos de poliformat se facilita un archivo powerpoint con los dibujos de los esquemas site-to-site y remote access para modificarlos y generar las imágenes.
- Configuración.

  Para la tarea 1: Explicar con capturas de pantalla y TEXTO EXPLICATIVO:

  - Órdenes utilizadas para configurar los tuns mostrando los resultados de las ejecuciones, comentando la utilidad.

- Tablas de routing (forwarding) explicando el resultado.

Para la tarea 2: Explicar con capturas de pantalla y TEXTO EXPLICATIVO:

- Órdenes utilizadas para activar el forwarding.
- Tablas de routing explicando el resultado.

- Funcionamiento del programa simpletun

  Realizar y mostrar las capturas de wireshark y TEXTO EXPLICATIVO que demuestre claramente el funcionamiento del túnel.

  Básicamente seguir los pasos que se indican en las tareas 1 y 2 a nivel de funcionamiento.

- Modificación de simpletun y funcionamiento del programa modificado (tarea 3 y tarea 4)

  - Realizar un listado del código fuente donde queden claras las modificaciones realizadas en ambas tareas y explicar estas insertando comentarios.
  - Realizar y mostrar las capturas de wireshark y TEXTO EXPLICATIVO que demuestre claramente el funcionamiento del túnel y del cifrado.

  Realizarlas usando la infraestructura de la tarea 2, monitorizando un ping (o un fichero pequeño de texto) entre los dos dd-wrt con las ips de su red local 10.24.x.x y 10.54.x.x.

- Conclusiones
- Referencias
- Apéndice con los listados

Consideraciones adicionales:

Las 4 máquinas virtuales deberán guardarse por si es necesario hacer alguna comprobación después de entregar el trabajo. El siguiente trabajo se realizará con una copia distinta de las máquinas virtuales.

El trabajo se entregará vía una tarea de poliformat en formato PDF y se incluirá un enlace de acceso de solo lectura al documento en overleaf.

El trabajo deberá entregarse antes de las 23:55 del martes 26 de noviembre.

Entrega fuera de plazo (a partir de las 23:56h) penalizarán en la nota >= 1pto

La nota de este trabajo aportará una puntuación del 25 % de la nota de la asignatura.

APÉNDICE A
Programación del tun/tap

Para abrir un interfaz tun o tap, hay que abrir el archivo /dev/net/tun y posteriormente, vía la llamada ioctl, seleccionar si es tun o tap y cual. Para eso hay que utilizar la estructura ifreq cuya definición se muestra en el listado A.1.

```
/*****************************************************************
 * Interface request structure used for socket ioctl's.
 * All interface ioctl's must have parameter definitions which
 * begin with ifr_name.
 * The remainder may be interface specific.
 *****************************************************************/
struct  ifreq {
#define IFNAMSIZ            16
    char        ifr_name[IFNAMSIZ];     /* if name, e.g. "tun0" */
    union {
        struct  sockaddr    ifru_addr;
        struct  sockaddr    ifru_dstaddr;
        struct  sockaddr    ifru_broadaddr;
        short       ifru_flags;
        int         ifru_metric;
        caddr_t     ifru_data;
    } ifr_ifru;
};
```

Listado A.1: La estructura ifreq

El nombre del interface (por ejemplo tun0 o tap3) se indicará en el campo ifr_name y además es necesario indicar el el campo ifru_flags una de estas dos constantes IFF_TUN o IFF_TAP en función de que se trate de un interfaz tun o tap. Opcionalmente podemos incluir (con una operación or) la constante IFF_NO_PI para indicar que no queremos que se incluya información de paquete (2 bytes para flags y 2 bytes para protocolo).

Nótese que en caso de no especificar el nombre, el sistema utilizará el primer interfaz disponible del tipo especificado en flags.

En simpletun.c la función que abre la interfaz tun o tap se denomina tun_alloc(). En el listado A.2 se incluye el código completo comentado.

```
/**************************************************************************
 * simpletun.c                                                            *
 *                                                                        *
 * A simplistic, simple-minded, naive tunnelling program using tun/tap    *
 * interfaces and TCP. Handles (badly) IPv4 for tun, ARP and IPv4 for     *
 * tap. DO NOT USE THIS PROGRAM FOR SERIOUS PURPOSES.                     *
 *                                                                        *
```

```
 * (C) 2009 Davide Brini.                                                  *
 *************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <linux/if.h>
#include <linux/if_tun.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <arpa/inet.h>
#include <sys/select.h>
#include <sys/time.h>
#include <errno.h>
#include <stdarg.h>

/* buffer for reading from tun/tap interface, must be >= 1500 */
#define BUFSIZE 2000
#define CLIENT 0
#define SERVER 1
#define PORT 55555

/* some common lengths */
#define IP_HDR_LEN 20
#define ETH_HDR_LEN 14
#define ARP_PKT_LEN 28

int debug;
char *progname;

/**************************************************************************
 * tun_alloc: allocates or reconnects to a tun/tap device. The caller     *
 *            needs to reserve enough space in *dev.                      *
 **************************************************************************/
int tun_alloc(char *dev, int flags) {

  struct ifreq ifr;
  int fd, err;

  /*---- abrimos /dev/net/tun ----*/
  if( (fd = open("/dev/net/tun", O_RDWR)) < 0 ) {
    perror("Opening /dev/net/tun");
    return fd;
  }

  /* ---- inicializamos a cero estructura ifreq ---- */
  memset(&ifr, 0, sizeof(ifr));

  /* ---- fijamos flags que nos pasan como parámetro (IFF_TUN o IFF_TAP) ----
     */
  ifr.ifr_flags = flags;

  /* ---- fijamos el nombre del interface que nos pasan como parámetro ---- */
  if (*dev) {
    strncpy(ifr.ifr_name, dev, IFNAMSIZ);
  }

  /* ---- llamamos a ioctl ---- */
  if( (err = ioctl(fd, TUNSETIFF, (void *)&ifr)) < 0 ) {
    perror("ioctl(TUNSETIFF)");
```

```
71      close(fd);
72      return err;
73    }
74
75    /* ---- si nos han pasado un nombre vacío, copiamos a dev el que ha utilizado
76            el sistema de forma automática ---- */
77    strcpy(dev, ifr.ifr_name);
78
79    /* ---- devolvemos el descriptor del archivo asociado al tun o tap ---- */
80    return fd;
81  }
82
83
84
85  /**************************************************************************
86   * cread: read routine that checks for errors and exits if an error is    *
87   *        returned.                                                       *
88   *************************************************************************/
89  int cread(int fd, char *buf, int n){
90
91    int nread;
92
93    if((nread=read(fd, buf, n))<0){
94      perror("Reading data");
95      exit(1);
96    }
97    return nread;
98  }
99
100 /**************************************************************************
101  * cwrite: write routine that checks for errors and exits if an error is  *
102  *         returned.                                                      *
103  *************************************************************************/
104 int cwrite(int fd, char *buf, int n){
105
106   int nwrite;
107
108   if((nwrite=write(fd, buf, n))<0){
109     perror("Writing data");
110     exit(1);
111   }
112   return nwrite;
113 }
114
115 /**************************************************************************
116  * read_n: ensures we read exactly n bytes, and puts those into "buf".    *
117  *         (unless EOF, of course)                                        *
118  *************************************************************************/
119 int read_n(int fd, char *buf, int n) {
120
121   int nread, left = n;
122
123   while(left > 0) {
124     if ((nread = cread(fd, buf, left))==0){
125       return 0 ;
126     }else {
127       left -= nread;
128       buf += nread;
129     }
130   }
131   return n;
132 }
133
134 /**************************************************************************
```

```
 * do_debug: prints debugging stuff (doh!)                                *
 ************************************************************************/
void do_debug(char *msg, ...){

  va_list argp;

  if(debug){
    va_start(argp, msg);
    vfprintf(stderr, msg, argp);
    va_end(argp);
  }
}

/**************************************************************************
 * my_err: prints custom error messages on stderr.                        *
 **************************************************************************/
void my_err(char *msg, ...) {

  va_list argp;

  va_start(argp, msg);
  vfprintf(stderr, msg, argp);
  va_end(argp);
}

/**************************************************************************
 * usage: prints usage and exits.                                         *
 **************************************************************************/
void usage(void) {
  fprintf(stderr, "Usage:\n");
  fprintf(stderr, "%s -i <ifacename> [-s|-c <serverIP>] [-p <port>] [-u|-a] [-d
      ]\n", progname);
  fprintf(stderr, "%s -h\n", progname);
  fprintf(stderr, "\n");
  fprintf(stderr, "-i <ifacename>: Name of interface to use (mandatory)\n");
  fprintf(stderr, "-s|-c <serverIP>: run in server mode (-s), or specify server
       address (-c <serverIP>) (mandatory)\n");
  fprintf(stderr, "-p <port>: port to listen on (if run in server mode) or to
      connect to (in client mode), default 55555\n");
  fprintf(stderr, "-u|-a: use TUN (-u, default) or TAP (-a)\n");
  fprintf(stderr, "-d: outputs debug information while running\n");
  fprintf(stderr, "-h: prints this help text\n");
  exit(1);
}

int main(int argc, char *argv[]) {

  int tap_fd, option;
  int flags = IFF_TUN;
  char if_name[IFNAMSIZ] = "";
  int header_len = IP_HDR_LEN;
  int maxfd;
  uint16_t nread, nwrite, plength;
//  uint16_t total_len, ethertype;
  char buffer[BUFSIZE];
  struct sockaddr_in local, remote;
  char remote_ip[16] = "";
  unsigned short int port = PORT;
  int sock_fd, net_fd, optval = 1;
  socklen_t remotelen;
  int cliserv = -1;    /* must be specified on cmd line */
  unsigned long int tap2net = 0, net2tap = 0;

  progname = argv[0];
```

```
196
197   /* Check command line options */
198   while((option = getopt(argc, argv, "i:sc:p:uahd")) > 0){
199     switch(option) {
200       case 'd':
201         debug = 1;
202         break;
203       case 'h':
204         usage();
205         break;
206       case 'i':
207         strncpy(if_name,optarg,IFNAMSIZ-1);
208         break;
209       case 's':
210         cliserv = SERVER;
211         break;
212       case 'c':
213         cliserv = CLIENT;
214         strncpy(remote_ip,optarg,15);
215         break;
216       case 'p':
217         port = atoi(optarg);
218         break;
219       case 'u':
220         flags = IFF_TUN;
221         break;
222       case 'a':
223         flags = IFF_TAP;
224         header_len = ETH_HDR_LEN;
225         break;
226       default:
227         my_err("Unknown option %c\n", option);
228         usage();
229     }
230   }
231
232   argv += optind;
233   argc -= optind;
234
235   if(argc > 0){
236     my_err("Too many options!\n");
237     usage();
238   }
239
240   if(*if_name == '\0'){
241     my_err("Must specify interface name!\n");
242     usage();
243   }else if(cliserv < 0){
244     my_err("Must specify client or server mode!\n");
245     usage();
246   }else if((cliserv == CLIENT)&&(*remote_ip == '\0')){
247     my_err("Must specify server address!\n");
248     usage();
249   }
250
251   /* initialize tun/tap interface */
252   if ( (tap_fd = tun_alloc(if_name, flags | IFF_NO_PI)) < 0 ) {
253     my_err("Error connecting to tun/tap interface %s!\n", if_name);
254     exit(1);
255   }
256
257   do_debug("Successfully connected to interface %s\n", if_name);
258
259   if ( (sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
```

```
260     perror("socket()");
261     exit(1);
262   }
263
264   /* ----------------------------------------------------
265   En esta parte conseguimos los dos descriptores asociados con los dos
266   interfaces de red que vamos a usar, el tun o tap en la variable tap_fd
267   y el socket al interfaz de red pública en la variable socket_fd.
268   El siguiente paso es inicializar el cliente o el servidor en función del
269   parámetro -s o -c que ya tenemos procesado en la variable cliserv.
270   Si es el cliente se conectará a la IP pública del servidor que tenemos en
271   la variable remote_ip:
272   ---------------------------------------------------- */
273
274   if(cliserv==CLIENT){
275     /* Client, try to connect to server */
276
277     /* assign the destination address */
278     memset(&remote, 0, sizeof(remote));
279     remote.sin_family = AF_INET;
280     remote.sin_addr.s_addr = inet_addr(remote_ip);
281     remote.sin_port = htons(port);
282
283     /* connection request */
284     if (connect(sock_fd, (struct sockaddr*) &remote, sizeof(remote)) < 0){
285       perror("connect()");
286       exit(1);
287     }
288
289     net_fd = sock_fd;
290     do_debug("CLIENT: Connected to server %s\n", inet_ntoa(remote.sin_addr));
291
292   } else {
293
294     /* Server, wait for connections
295     Si es el servidor, reservaremos el puerto correspondiente con bind y
296     quedaremos en espera con listen + accept hasta que se conecte el cliente:
297      */
298
299     /* avoid EADDRINUSE error on bind() */
300     if(setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, (char *)&optval, sizeof(
          optval)) < 0){
301       perror("setsockopt()");
302       exit(1);
303     }
304
305     memset(&local, 0, sizeof(local));
306     local.sin_family = AF_INET;
307     local.sin_addr.s_addr = htonl(INADDR_ANY);
308     local.sin_port = htons(port);
309     if (bind(sock_fd, (struct sockaddr*) &local, sizeof(local)) < 0){
310       perror("bind()");
311       exit(1);
312     }
313
314     if (listen(sock_fd, 5) < 0){
315       perror("listen()");
316       exit(1);
317     }
318
319     /* wait for connection request */
320     remotelen = sizeof(remote);
321     memset(&remote, 0, remotelen);
322     if ((net_fd = accept(sock_fd, (struct sockaddr*)&remote, &remotelen)) < 0){
```

```
323       perror("accept()");
324       exit(1);
325     }
326
327     do_debug("SERVER: Client connected from %s\n", inet_ntoa(remote.sin_addr));
328   }
329
330
331
332   /*
333   Después de la conexión en ambos casos utilizaremos la variable net_fd para
334   acceder a los datos que vengan de la red pública que conecta cliente con
335   servidor.
336   Después de que se hayan establecido las conexiones, tanto el cliente como el
337   servidor hacen exactamente lo mismo, bucle en el que esperan actividad de
338   cualquiera de las dos fuentes (tun/tap o net) y retransmiten lo recibido por
339   la otra fuente.
340   Se utiliza la sentencia select para esperar a que llegue un paquete.
341   Cuando se reciben n bytes desde tun/tap se envía por el interfaz net, primero
342   la longitud de lo recibido y luego los datos.
343   Cuando se recibe un paquete de net, se lee en primer lugar la longitud y a
344   continuación se lee los bytes indicados por dicha longitud.
345   */
346
347   /* use select() to handle two descriptors at once */
348   maxfd = (tap_fd > net_fd)?tap_fd:net_fd;
349
350   while(1) {
351     int ret;
352     fd_set rd_set;
353
354     FD_ZERO(&rd_set);
355     FD_SET(tap_fd, &rd_set); FD_SET(net_fd, &rd_set);
356
357     ret = select(maxfd + 1, &rd_set, NULL, NULL, NULL);
358
359     if (ret < 0 && errno == EINTR){
360       continue;
361     }
362
363     if (ret < 0) {
364       perror("select()");
365       exit(1);
366     }
367
368     if(FD_ISSET(tap_fd, &rd_set)){
369       /* data from tun/tap: just read it and write it to the network */
370
371       nread = cread(tap_fd, buffer, BUFSIZE);
372
373       tap2net++;
374       do_debug("TAP2NET %lu: Read %d bytes from the tap interface\n", tap2net,
              nread);
375
376       /*
377       --->
378       --->  Posible punto de inserción de código usuario
379       --->
380       */
381
382       /* write length + packet */
383       plength = htons(nread);
384       nwrite = cwrite(net_fd, (char *)&plength, sizeof(plength));
385       nwrite = cwrite(net_fd, buffer, nread);
```

```
386
387         do_debug("TAP2NET %lu: Written %d bytes to the network\n", tap2net,
                nwrite);
388       }
389
390       if(FD_ISSET(net_fd, &rd_set)){
391         /* data from the network: read it, and write it to the tun/tap interface.
392          * We need to read the length first, and then the packet */
393
394         /* Read length */
395         nread = read_n(net_fd, (char *)&plength, sizeof(plength));
396         if(nread == 0) {
397           /* ctrl-c at the other end */
398           break;
399         }
400
401         net2tap++;
402
403         /* read packet */
404         nread = read_n(net_fd, buffer, ntohs(plength));
405         do_debug("NET2TAP %lu: Read %d bytes from the network\n", net2tap, nread)
                ;
406
407         /* now buffer[] contains a full packet or frame, write it into the tun/
                tap interface */
408
409         /*
410         --->
411         --->  Posible punto de inserción de código usuario
412         --->
413         */
414
415         nwrite = cwrite(tap_fd, buffer, nread);
416         do_debug("NET2TAP %lu: Written %d bytes to the tap interface\n", net2tap,
                nwrite);
417       }
418     }
419
420     return(0);
421 }
```

Listado A.2: Código fuente de simpletun.c