



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Trabajo 3: VPN con “simpletun”

PRÁCTICA RCO

Grado en Ingeniería Informática

Autor: Diego Córdoba Serra
Javier García Bartolomé
Maria Carmen Rea Mejia
Grupo: 181

Curso 2024-2025

Resumen

En este trabajo, se analizará el funcionamiento de una VPN utilizando simpletun, un programa que puede operar tanto como cliente como servidor y que establece un túnel TCP entre ambos.

Para ello, en primer lugar, se procederá a realizar la configuración necesaria sobre la red virtual, lo que incluirá la configuración de la redirección de puertos y la preparación del sistema operativo Linux para el manejo de las interfaces tun/tap. Se seleccionará el dispositivo RCO-noX como servidor simpletun, mientras que RCO-X actuará como cliente, estableciendo la conexión con el servidor a través de la IP de la WAN del router ddwrt-nox.

Finalmente, se llevarán a cabo una serie de tareas prácticas que permitirán profundizar en el entendimiento del funcionamiento de una VPN utilizando simpletun, proporcionando una visión clara de su implementación y operatividad. Entre estas tareas podemos destacar la modificación del código fuente de simpletun para hacer uso de cifrados simples, en concreto los cifrados Caesar y XOR, que nos permitirá dotar al túnel de una mayor seguridad.

Palabras clave: Redes Virtuales, Túnel TUN/TAP, Simpletun, VPN, Cifrado, Caesar, XOR, Máquinas Virtuales, DD-WRT, AlmaLinux, Cliente-Servidor

Abstract

In this work, we will analyze the operation of a VPN using simpletun, a program that can operate as both a client and a server and that establishes a TCP tunnel between them.

To do this, we will first proceed to perform the necessary configuration on the virtual network, which will include the configuration of port forwarding and the preparation of the Linux operating system to handle the tun/tap interfaces. The RCO-noX device will be selected as the simpletun server, while RCO-X will act as the client, establishing the connection with the server through the WAN IP of the ddwrt-nox router.

Finally, we will carry out a series of practical tasks that will allow us to deepen our understanding of the operation of a VPN using simpletun, providing a clear vision of its implementation and operation. Among these tasks we can highlight the modification of the simpletun source code to make use of simple ciphers, specifically Caesar and XOR ciphers, which will allow us to provide the tunnel with greater security.

Key words: Virtual Networks, TUN/TAP Tunnel, Simpletun, VPN, Encryption, Caesar, XOR, Virtual Machines, DD-WRT, AlmaLinux, Client-Server

Índice general

Índice general	V
Índice de figuras	VII
Índice de Listados	VIII

1 Introducción	1
1.1 Objetivos	2
2 Configuración	3
2.1 Configuración de la redirección de puertos	3
2.2 Preparación de CentOS para trabajar con tun/tap	4
2.3 Configuración de las interfaces y activación del túnel simpletun	5
2.3.1 Script servidor RCO-noX	5
2.3.2 Script cliente RCO-X	5
3 Funcionamiento del programa simpletun	7
3.1 Tarea-1: Comprobación del túnel	7
3.2 Tarea-2: Túnel como site-to-site	12
4 Modificación de simpletun y funcionamiento del programa modificado	17
4.1 Tarea-3: Cifrado Caesar	17
4.1.1 Implementación	18
4.1.2 Comprobación	19
4.2 Tarea-4: Cifrado 'secreto'	20
4.2.1 Implementación	21
4.2.2 Comprobación	22
5 Conclusiones	25
Bibliografía	27

Apéndice	
A Listados	29

Índice de figuras

1.1	Esquema de red	1
2.1	Regla de port forwarding en ddwrt-noX	3
2.2	Orden iptables en ddwrt-noX	4
2.3	gcc RCO-noX	4
2.4	gcc RCO-X	5
2.5	Enter Caption	5
2.6	Enter Caption	5
3.1	ping desde ROC-X a 10.24.181.1 por tunel	7
3.2	wireshark ping tunel	8
3.3	datagrama ICMP	8
3.4	datagrama TCP	9
3.5	Campo data del ICMP	9
3.6	Campo data del tercer TCP PSH/ACK	9
3.7	ping desde ROC-X a 10.24.181.2 por tunel	10
3.8	Captura con wireshark del ping desde RCO-X a 10.24.181.2	10
3.9	ping desde ROC-X a 10.24.181.2 por tunel, reglas cambiadas	11
3.10	direcciones con reglas cambiadas	11
3.11	Nueva regla routing ddwrt-noX	12
3.12	route -n ddwrt-noX	12
3.13	Nueva regla routing anfitrion	13
3.14	Nueva regla routing ddwrt-X	13
3.15	comunican con 10.54, ping ddwrt-X a ddwrt-noX	14
3.16	Enter Caption	14
3.17	Wireshark ping ddwrt-noX a ddwrt-X	14
3.18	ICMP cabecera Ethernet	15
3.19	ICMP sin cabecera Ethernet	15
4.1	Ejemplo de cifrado Caesar	17
4.2	Variables añadidas	18
4.3	Codificación Caesar	18
4.4	Descodificación Caesar	19
4.5	Ping para ver el funcionamiento	19
4.6	Ping con cifrado Caesar(I)	20
4.7	Ping con cifrado Caesar(II)	20
4.8	ping cifrado secreto	22
4.9	Mensaje sin cifrado secreto	23
4.10	mensaje con cifrado secreto	23

Índice de Listados

A.1	codigo fuente de simpletun.c	29
-----	--	----

CAPÍTULO 1

Introducción

En este trabajo se establece una conexión entre máquinas virtuales con un ordenador anfitrión, mediante una red virtual privada.

Para ello, se hace uso de los protocolos Tun y Tap, configurando así un túnel a través del programa `simpletun`, que gestiona el tráfico al leer los datos provenientes del Tun y lo escribe por la red y viceversa.

La diferencia entre una interfaz Tap y una interfaz Tun radica en el tipo de datos que manejan: Tap gestiona tramas Ethernet, mientras que Tun trabaja con datagramas IP. La interfaz Tap es ideal para configurar puentes, mientras que Tun se utiliza para establecer túneles.

Partimos de la configuración previa empleada en trabajos anteriores, compuesta por cuatro máquinas virtuales y un ordenador anfitrión. Como podemos observar en la siguiente Figura 1.1, tenemos dos routers de tipo `ddwrt`, que conecta a la máquina `RCO-noX` con el router `ddwrt-noX` y el otro conecta a la máquina `RCO-X` con el router `ddwrt-X`. Cabe destacar que tanto el `RCO-noX` (actuará como servidor) como el `RCO-X` (actuará como cliente) cuentan con un sistema operativo AlmaLinux, y `RCO-X` se conectará al servidor usando la IP de la WAN del router `ddwrt-noX`.

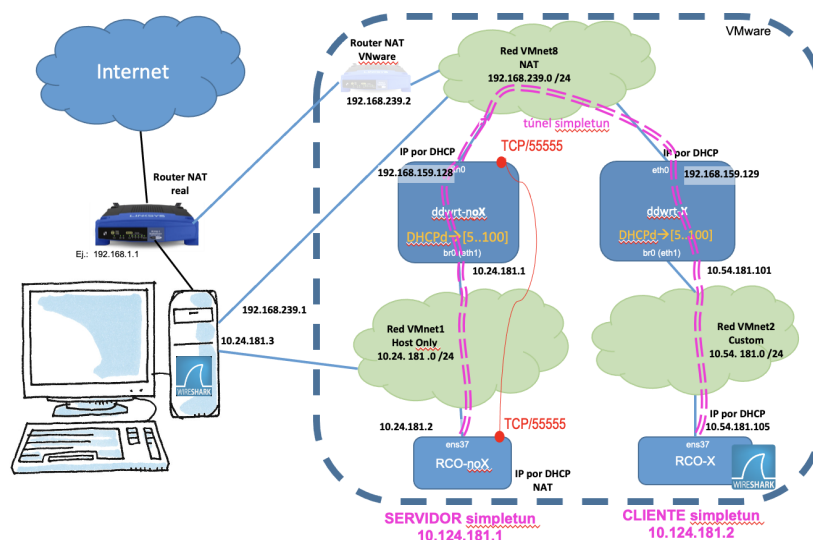


Figura 1.1: Esquema de red

1.1 Objetivos

El objetivo principal es conectar las máquinas de redes diferentes mediante una red virtual privada, y así poder lograr la implementación correcta de un túnel tun/tap mediante el uso de simpletun. Esto incluye configurar máquinas virtuales específicas, establecer conexiones entre ellas, analizar y realizar varias pruebas, y así poder documentar de forma detallada el proceso.

Además, se busca experimentar con la integración de métodos de cifrado mediante el código de simpletun.

CAPÍTULO 2

Configuración

Como se mencionó antes, vamos a partir del entregable PPTP y modificaremos algunas configuraciones de las máquinas, como la desactivación del PPTP, las rutas IP y las redirecciones de puertos. Además, configuraremos RCO-noX como un servidor simpletun y RCO-X actuará como cliente y se conectará al servidor usando la IP de la WAN del router ddwrt-noX.

2.1 Configuración de la redirección de puertos

Primero partimos del router ddwrt-noX, para ello accederemos desde nuestro navegador a la dirección IP WAN (192.168.159.128) y seleccionaremos NAT/QoS >Port Forwarding. A continuación, pulsaremos Add para así añadir nuestra regla de forwarding, en nuestro caso le llamaremos simpletun. Como podemos observar en la Figura 2.1, debemos de indicar que el tráfico que entre por el puerto TCP 55555 se dirija a este mismo puerto de RCO-noX (dirección IP 10.24.181.2).

Después de realizar esta configuración, hacemos clic en 'Apply Settings'.

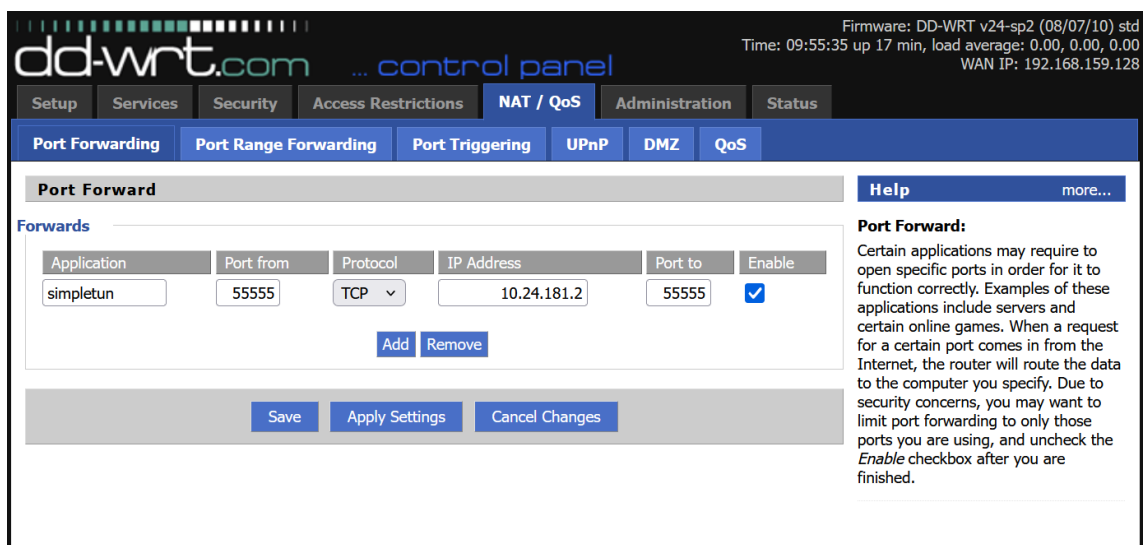


Figura 2.1: Regla de port forwarding en ddwrt-noX

Una vez aplicada la configuración, podemos acceder al terminal del router y ejecutar la orden iptables para verificar que la redirección se ha creado correctamente.

En la Figura 2.2 se puede observar una nueva línea que indica que los paquetes TCP que lleguen al puerto 55555 del router serán redirigidos a la dirección IP 10.24.181.2, también por el puerto 55555.

```
root@DD-WRT:~# iptables -nL -t nat
Chain PREROUTING (policy ACCEPT)
target     prot opt source                destination              tcp dpt:8080 to:10.24.181.1:80
DNAT       tcp  --  0.0.0.0/0              192.168.159.128          tcp dpt:22 to:10.24.181.1:22
DNAT       tcp  --  0.0.0.0/0              192.168.159.128          to:10.24.181.1
DNAT       icmp --  0.0.0.0/0              192.168.159.128          tcp dpt:55555 to:10.24.181.2:55555 ←
TRIGGER    0    --  0.0.0.0/0              192.168.159.128          TRIGGER type:dnat match:0 relate:0

Chain POSTROUTING (policy ACCEPT)
target     prot opt source                destination              to:192.168.159.128
SNAT       0    --  0.0.0.0/0              0.0.0.0/0                PKTTYPE = broadcast
RETURN     0    --  0.0.0.0/0              0.0.0.0/0
MASQUERADE 0    --  10.24.181.0/24          10.24.181.0/24

Chain OUTPUT (policy ACCEPT)
target     prot opt source                destination
root@DD-WRT:~#
```

Figura 2.2: Orden iptables en ddwrt-noX

2.2 Preparación de CentOS para trabajar con tun/tap

Para trabajar con TUN/TAP, es necesario instalar gcc, el compilador de c y c++, en las máquinas RCO. Para ello, abrimos un terminal en ambos hosts y ejecutamos el siguiente comando:

```
1 # yum install gcc
```

Por otro lado, necesitamos descargar el archivo simpletun.c desde la web oficial 'https://web.ecs.syr.edu/~wedu/seed/Labs/VPN/files/simpletun.c' y compilarlo:

```
1 # cd
2 # mkdir simpletun
3 # cd simpletun/
4 # wget https://redescorporativas.es/simpletun.c
5 # make simpletun
```

Con esto ya tenemos instalado gcc en la máquina RCO-noX Figura 2.3 y RCO-X Figura 2.4:

```
[root@rco-nox ~]# gcc --version
gcc (GCC) 8.5.0 20210514 (Red Hat 8.5.0-22)
Copyright (C) 2018 Free Software Foundation, Inc.
Esto es software libre; vea el código para las condiciones de copia. NO hay
garantía; ni siquiera para MERCANTIBILIDAD o IDONEIDAD PARA UN PROPÓSITO EN
PARTICULAR

[root@rco-nox ~]#
```

Figura 2.3: gcc RCO-noX

```
[root@rco-x ~]# gcc --version
gcc (GCC) 8.5.0 20210514 (Red Hat 8.5.0-22)
Copyright (C) 2018 Free Software Foundation, Inc.
Esto es software libre; vea el código para las condiciones de copia. NO hay
garantía; ni siquiera para MERCANTIBILIDAD o IDONEIDAD PARA UN PROPÓSITO EN
PARTICULAR
[root@rco-x ~]#
```

Figura 2.4: gcc RCO-X

2.3 Configuración de las interfaces y activación del túnel simpletun

Hemos preparado un script en cada máquina para crear la configuración del túnel.

Por otro lado, se ha creado un fichero de texto 'help' que nos recuerda, en cada etapa del trabajo, el orden de ejecución de los comandos para su correcto inicio.

2.3.1. Script servidor RCO-noX

```
1 #!/bin/bash
2 ip tuntap add dev tun0 mode tun
3 ip link set tun0 up
4 ip addr add 10.124.181.1/30 dev tun0
```

```
[root@rco-nox simpletun]# cat help
- Ejecutar script.sh
- Ejecutar:
    ./simpletun -i tun0 -s &
- Tras haber hecho lo mismo en la otra maquina, ejecutar:
    ip route add 10.54.181.0/24 dev tun0
```

Figura 2.5: Enter Caption

2.3.2. Script cliente RCO-X

```
1 #!/bin/bash
2 ip tuntap add dev tun3 mode tun
3 ip link set tun3 up
4 ip addr add 10.124.181.2/30 dev tun3
```

```
[root@rco-x simpletun]# cat help
- Ejecutar script.sh
- Ejecutar:
    ./simpletun -i tun3 -c 192.168.159.128 &
- Tras haber hecho lo mismo en la otra maquina, ejecutar:
    ip route add 10.24.181.0/24 dev tun3
```

Figura 2.6: Enter Caption

CAPÍTULO 3

Funcionamiento del programa simpletun

3.1 Tarea-1: Comprobación del túnel

Se procederá a realizar una comprobación del túnel para verificar su funcionamiento mediante la captura y análisis del tráfico que muestra el encapsulado del túnel, utilizando la herramienta Wireshark y el comando ping. El objetivo es observar el tráfico encapsulado dentro del túnel y explicar el funcionamiento de simpletun a partir de los datos obtenidos.

En primer lugar, se capturará el tráfico de los adaptadores ens37 y tun3 en la máquina RCO-X utilizando Wireshark. Una vez configurada la captura de tráfico, se procederá a realizar un ping desde RCO-X a la dirección IP 10.124.181.1. Para facilitar la identificación de los paquetes ICMP en la captura, se especificará un tamaño de paquete de 1000 bytes mediante el comando ping -s 1000 10.124.181.1.

```
[root@rco-x simpletun]# ping -s 1000 10.124.181.1
PING 10.124.181.1 (10.124.181.1) 1000(1028) bytes of data.
1008 bytes from 10.124.181.1: icmp_seq=1 ttl=64 time=2.69 ms
1008 bytes from 10.124.181.1: icmp_seq=2 ttl=64 time=43.8 ms
1008 bytes from 10.124.181.1: icmp_seq=3 ttl=64 time=43.8 ms
1008 bytes from 10.124.181.1: icmp_seq=4 ttl=64 time=43.8 ms
1008 bytes from 10.124.181.1: icmp_seq=5 ttl=64 time=42.7 ms
1008 bytes from 10.124.181.1: icmp_seq=6 ttl=64 time=43.2 ms
1008 bytes from 10.124.181.1: icmp_seq=7 ttl=64 time=43.8 ms
1008 bytes from 10.124.181.1: icmp_seq=8 ttl=64 time=43.7 ms
^C
--- 10.124.181.1 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 7013ms
rtt min/avg/max/mdev = 2.689/38.428/43.802/13.514 ms
[root@rco-x simpletun]#
```

Figura 3.1: ping desde ROC-X a 10.24.181.1 por tunel

Ahora detendremos la captura en wireshark y podremos ver por cada ping 1 ICMP echo request, 8 TCPs y 1 ICMP echo reply.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.124.181.2	10.124.181.1	ICMP	1028	Echo (ping) request id=0x0004, seq=1/256, ttl=64 (repl...
3	0.000092387	10.54.181.105	192.168.159.128	TCP	68	36360 → 55555 [PSH, ACK] Seq=1 Ack=1 Win=735 Len=2 TSva...
4	0.001044808	192.168.159.128	10.54.181.105	TCP	66	55555 → 36360 [ACK] Seq=1 Ack=3 Win=725 Len=0 TSval=104...
5	0.001065458	10.54.181.105	192.168.159.128	TCP	1094	36360 → 55555 [PSH, ACK] Seq=3 Ack=1 Win=735 Len=1028 T...
6	0.001837549	192.168.159.128	10.54.181.105	TCP	66	55555 → 36360 [ACK] Seq=1 Ack=1031 Win=741 Len=0 TSval=...
7	0.001929023	192.168.159.128	10.54.181.105	TCP	68	55555 → 36360 [PSH, ACK] Seq=1 Ack=1031 Win=741 Len=2 T...
8	0.001939951	10.54.181.105	192.168.159.128	TCP	66	36360 → 55555 [ACK] Seq=1031 Ack=3 Win=735 Len=0 TSval=...
9	0.002604892	192.168.159.128	10.54.181.105	TCP	1094	55555 → 36360 [PSH, ACK] Seq=3 Ack=1031 Win=741 Len=102...
10	0.002615347	10.54.181.105	192.168.159.128	TCP	66	36360 → 55555 [ACK] Seq=1031 Ack=1031 Win=751 Len=0 TSV...
2	0.002664081	10.124.181.1	10.124.181.2	ICMP	1028	Echo (ping) reply id=0x0004, seq=1/256, ttl=64 (requ...
11	1.001529618	10.124.181.2	10.124.181.1	ICMP	1028	Echo (ping) request id=0x0004, seq=2/512, ttl=64 (repl...
13	1.001626455	10.54.181.105	192.168.159.128	TCP	68	36360 → 55555 [PSH, ACK] Seq=1031 Ack=1031 Win=751 Len=...
14	1.043621412	192.168.159.128	10.54.181.105	TCP	66	55555 → 36360 [ACK] Seq=1031 Ack=1033 Win=741 Len=0 TSV...
15	1.043656030	10.54.181.105	192.168.159.128	TCP	1094	36360 → 55555 [PSH, ACK] Seq=1033 Ack=1031 Win=751 Len=...

Figura 3.2: wireshark ping tunel

Ahora nos dispondremos a analizar detenidamente que significa y porqué aparecen cada uno de estos protocolos en la captura.

Empezando por el paquete ICMP echo request, este paquete es enviado por el dispositivo que inicia el ping (RCO-X) hacia el destino (servidor simpletun). Este paquete contiene una solicitud para que el dispositivo de destino responda, y su propósito es verificar la conectividad.

Respecto al paquete ICMP Echo Reply, es una respuesta generada por el dispositivo de destino al recibir un ICMP Echo Request. Su propósito es confirmar que el dispositivo de destino ha recibido correctamente el paquete de solicitud y está disponible para la comunicación. La recepción del ICMP Echo Reply por parte del dispositivo de origen indica que el dispositivo de destino está operativo y accesible en la red.

La razón por la cual ambos paquetes no presentan cabecera Ethernet es que el ping se ha enviado al servidor simpletun, lo que implica que la interfaz que captura los paquetes ICMP es tun3, una interfaz tunelizada (TUN/TAP). Este tipo de interfaces operan en la capa 3 del modelo OSI, es decir, en la capa de red (IP), y no procesan tramas de nivel 2 como las asociadas con Ethernet. Debido a esta característica, las interfaces TUN/TAP no incluyen cabeceras Ethernet en los paquetes que gestionan, lo que explica la ausencia de estas cabeceras en los paquetes ICMP capturados.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.124.181.2	10.124.181.1	ICMP	1028	Echo (ping) request id=0x0004, seq=1/256, ttl=64 (rep...
3	0.000092387	10.54.181.105	192.168.159.128	TCP	68	36360 → 55555 [PSH, ACK] Seq=1 Ack=1 Win=735 Len=2 TSva...
4	0.001044808	192.168.159.128	10.54.181.105	TCP	66	55555 → 36360 [ACK] Seq=1 Ack=3 Win=725 Len=0 TSval=104...

Frame 1: 1028 bytes on wire (8224 bits), 1028 bytes captured (8224 bits) on interface 0

Raw packet data

Internet Protocol Version 4, Src: 10.124.181.2, Dst: 10.124.181.1

Internet Control Message Protocol

Figura 3.3: datagrama ICMP

Nos centramos ahora en los paquetes TCP. Los dos primeros paquetes corresponden al establecimiento de la conexión, siendo el primero el que inicia la comunicación, mientras que el segundo actúa como un ACK. Es en el tercer paquete donde se efectúa la transmisión de datos, observándose que en cada paquete enviado se establece el flag PSH (Push) para indicar la transmisión de datos, seguido inmediatamente por su correspondiente ACK (flag ACK), que confirma la recepción de dichos datos. En los cuatro últimos paquetes TCP podemos observar que las ips source y destination se han invertido, lo que significa que estos mensajes son respuestas del servidor y sus correspondientes ACKs.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.124.181.2	10.124.181.1	ICMP	1028	Echo (ping) request id=0x0004, seq=1/256, ttl=64 (reply in 2)
3	0.000092387	10.54.181.105	192.168.159.128	TCP	68	36360 → 55555 [PSH, ACK] Seq=1 Ack=1 Win=735 Len=2 TSval=1045503982 TSecr=1045503982
4	0.001044808	192.168.159.128	10.54.181.105	TCP	66	55555 → 36360 [ACK] Seq=1 Ack=3 Win=725 Len=0 TSval=1045503982 TSecr=1045503982
Frame 3: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface 1 Ethernet II, Src: Vmware_38:60:9c (00:50:56:38:60:9c), Dst: Vmware_3e:ab:c9 (00:50:56:3e:ab:c9) Internet Protocol Version 4, Src: 10.54.181.105, Dst: 192.168.159.128 Transmission Control Protocol, Src Port: 36360, Dst Port: 55555, Seq: 1, Ack: 1, Len: 2 Data (2 bytes)						

Figura 3.4: datagrama TCP

Como podemos observar verificando los valores hexadecimales del campos de datos en las figuras 3.3 y 3.4, el campo de datos del segmento TCP coincide con el datagrama IP del ICMP echo correspondiente. Esto confirma que los datos del ICMP están encapsulados dentro del segmento TCP.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.124.181.2	10.124.181.1	ICMP	1028	Echo (ping) request id=0x0004, seq=1/256, ttl=64 (reply in 2)
3	0.000092387	10.54.181.105	192.168.159.128	TCP	68	36360 → 55555 [PSH, ACK] Seq=1 Ack=1 Win=735 Len=2 TSval=1045503982 TSecr=1045503982
4	0.001044808	192.168.159.128	10.54.181.105	TCP	66	55555 → 36360 [ACK] Seq=1 Ack=3 Win=725 Len=0 TSval=1045503982 TSecr=1045503982
5	0.001065458	10.54.181.105	192.168.159.128	TCP	1094	36360 → 55555 [PSH, ACK] Seq=3 Ack=1 Win=735 Len=1028 TSval=1045503982 TSecr=1045503982
6	0.001837549	192.168.159.128	10.54.181.105	TCP	66	55555 → 36360 [ACK] Seq=1 Ack=1031 Win=741 Len=0 TSval=1045503983 TSecr=1045503983
7	0.001929023	192.168.159.128	10.54.181.105	TCP	68	55555 → 36360 [PSH, ACK] Seq=1 Ack=1031 Win=741 Len=2 TSval=1045503983 TSecr=1045503983
8	0.001939951	10.54.181.105	192.168.159.128	TCP	66	36360 → 55555 [ACK] Seq=1031 Ack=3 Win=735 Len=0 TSval=1045503983 TSecr=1045503983
9	0.002604892	192.168.159.128	10.54.181.105	TCP	1094	55555 → 36360 [PSH, ACK] Seq=3 Ack=1031 Win=741 Len=1028 TSval=1045503984 TSecr=1045503984
Checksum: 0x5636 [correct] [Checksum Status: Good] Identifier (BE): 4 (0x0004) Identifier (LE): 1024 (0x0400) Sequence number (BE): 1 (0x0001) Sequence number (LE): 256 (0x0100) [Response frame: 2] Timestamp from icmp data: Nov 27, 2024 22:48:05.000000000 CET [Timestamp from icmp data (relative): 0.493181051 seconds]						
Data (992 bytes) Data: 6486070000000000101112131415161718191a1b1c1d1e1f... [Length: 992]						
0020	00 00 00 00 04 06 07 00	00 00 00 00 10 11 12 13
0030	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0040	24 25 26 27 28 29 2a 2b	2c 2d 2e 2f 30 31 32 33
0050	34 35 36 37 38 39 3a 3b	3c 3d 3e 3f 40 41 42 43
0060	44 45 46 47 48 49 4a 4b	4c 4d 4e 4f 50 51 52 53
0070	54 55 56 57 58 59 5a 5b	5c 5d 5e 5f 60 61 62 63
0080	64 65 66 67 68 69 6a 6b	6c 6d 6e 6f 70 71 72 73
0090	74 75 76 77 78 79 7a 7b	7c 7d 7e 7f 80 81 82 83

Figura 3.5: Campo data del ICMP

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.124.181.2	10.124.181.1	ICMP	1028	Echo (ping) request id=0x0004, seq=1/256, ttl=64 (reply in 2)
3	0.000092387	10.54.181.105	192.168.159.128	TCP	68	36360 → 55555 [PSH, ACK] Seq=1 Ack=1 Win=735 Len=2 TSval=1045503982 TSecr=1045503982
4	0.001044808	192.168.159.128	10.54.181.105	TCP	66	55555 → 36360 [ACK] Seq=1 Ack=3 Win=725 Len=0 TSval=1045503982 TSecr=1045503982
5	0.001065458	10.54.181.105	192.168.159.128	TCP	1094	36360 → 55555 [PSH, ACK] Seq=3 Ack=1 Win=735 Len=1028 TSval=1045503982 TSecr=1045503982
6	0.001837549	192.168.159.128	10.54.181.105	TCP	66	55555 → 36360 [ACK] Seq=1 Ack=1031 Win=741 Len=0 TSval=1045503983 TSecr=1045503983
7	0.001929023	192.168.159.128	10.54.181.105	TCP	68	55555 → 36360 [PSH, ACK] Seq=1 Ack=1031 Win=741 Len=2 TSval=1045503983 TSecr=1045503983
8	0.001939951	10.54.181.105	192.168.159.128	TCP	66	36360 → 55555 [ACK] Seq=1031 Ack=3 Win=735 Len=0 TSval=1045503983 TSecr=1045503983
9	0.002604892	192.168.159.128	10.54.181.105	TCP	1094	55555 → 36360 [PSH, ACK] Seq=3 Ack=1031 Win=741 Len=1028 TSval=1045503984 TSecr=1045503984
...0. = More fragments: Not set ...0 0000 0000 0000 = Fragment offset: 0 Time to live: 64 Protocol: TCP (6) Header checksum: 0xfbb3 [validation disabled] [Header checksum status: Unverified] Source: 10.54.181.105 Destination: 192.168.159.128 Transmission Control Protocol, Src Port: 36360, Dst Port: 55555, Seq: 3, Ack: 1, Len: 1028						
Data (1028 bytes) Data: 45080404202400040012520a7cb50108005636... [Length: 1028]						
0040	1f ee 15 00 04 04 92 d2	40 00 40 01 25 20 0a 7c
0050	05 02 0a 7c b5 01 08 00	56 36 00 04 00 01 95 93
0060	47 07 00 00 00 00 04 06	07 00 00 00 00 00 10 11
0070	12 13 14 15 16 17 18 19	1a 1b 1c 1d 1e 1f 20 21
0080	22 23 24 25 26 27 28 29	2a 2b 2c 2d 2e 2f 30 31
0090	32 33 34 35 36 37 38 39	3a 3b 3c 3d 3e 3f 40 41
00a0	42 43 44 45 46 47 48 49	4a 4b 4c 4d 4e 4f 50 51
00b0	52 53 54 55 56 57 58 59	5a 5b 5c 5d 5e 5f 60 61
00c0	62 63 64 65 66 67 68 69	6a 6b 6c 6d 6e 6f 70 71
00d0	72 73 74 75 76 77 78 79	7a 7b 7c 7d 7e 7f 80 81
00e0	82 83 84 85 86 87 88 89	8a 8b 8c 8d 8e 8f 90 91
00f0	92 93 94 95 96 97 98 99	9a 9b 9c 9d 9e 9f a0 a1
0100	a2 a3 a4 a5 a6 a7 a8 a9	aa ab ac ad ae af b0 b1
0110	b2 b3 b4 b5 b6 b7 b8 b9	ba bb bc bd be bf c0 c1

Figura 3.6: Campo data del tercer TCP PSH/ACK

Vamos a verificar que la regla de routing añadida funciona correctamente, para ello realizaremos desde RCO-X un ping -s 1000 10.24.181.2.

```
[root@rc0-x simpletun]# ping -s 1000 10.24.181.2
PING 10.24.181.2 (10.24.181.2) 1000(1028) bytes of data.
1008 bytes from 10.24.181.2: icmp_seq=1 ttl=64 time=44.1 ms
1008 bytes from 10.24.181.2: icmp_seq=2 ttl=64 time=43.2 ms
1008 bytes from 10.24.181.2: icmp_seq=3 ttl=64 time=43.6 ms
1008 bytes from 10.24.181.2: icmp_seq=4 ttl=64 time=42.9 ms
^C
--- 10.24.181.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 42.944/43.464/44.071/0.470 ms
[root@rc0-x simpletun]#
```

Figura 3.7: ping desde ROC-X a 10.24.181.2 por tunel

Observamos con wireshark las direcciones fuente y destino.

No.	Time	Source	Destination	Protocol	Length	Info
42	3.047819509	10.24.181.2	10.124.181.2	ICMP	1028	Echo (ping) reply id=0x0007, seq=4/1024, ttl=64 (request in 33)
41	3.047731565	10.54.181.105	192.168.159.128	TCP	60	36360 → 55555 [ACK] Seq=4121 Ack=4121 Win=1056 Len=0 TSval=3191813882 TSecr=1048393311
40	3.047721426	192.168.159.128	10.54.181.105	TCP	1094	55555 → 36360 [PSH, ACK] Seq=3093 Ack=4121 Win=1046 Len=1028 TSval=1048393311 TSecr=3191813881
39	3.046953177	10.54.181.105	192.168.159.128	TCP	60	36360 → 55555 [ACK] Seq=4121 Ack=3093 Win=1049 Len=0 TSval=3191813881 TSecr=1048393311
38	3.046941567	192.168.159.128	10.54.181.105	TCP	60	55555 → 36360 [PSH, ACK] Seq=3091 Ack=4121 Win=1046 Len=2 TSval=1048393311 TSecr=3191813880
37	3.046895249	192.168.159.128	10.54.181.105	TCP	60	55555 → 36360 [ACK] Seq=3091 Ack=4121 Win=1046 Len=0 TSval=1048393311 TSecr=3191813880
36	3.046850450	10.54.181.105	192.168.159.128	TCP	1094	36360 → 55555 [PSH, ACK] Seq=3093 Ack=3091 Win=1040 Len=1028 TSval=3191813880 TSecr=1048393310
35	3.046823411	192.168.159.128	10.54.181.105	TCP	60	55555 → 36360 [ACK] Seq=3091 Ack=3093 Win=1030 Len=0 TSval=1048393310 TSecr=3191813839
34	3.046833430	10.54.181.105	192.168.159.128	TCP	60	36360 → 55555 [PSH, ACK] Seq=3091 Ack=3091 Win=1040 Len=2 TSval=3191813839 TSecr=1048392310
33	3.046899197	10.124.181.2	10.24.181.2	ICMP	1028	Echo (ping) request id=0x0007, seq=4/1024, ttl=64 (reply in 42)
25	2.046566679	10.24.181.2	10.124.181.2	ICMP	1028	Echo (ping) reply id=0x0007, seq=3/768, ttl=64 (request in 23)
32	2.046496436	10.54.181.105	192.168.159.128	TCP	60	36360 → 55555 [ACK] Seq=3091 Ack=3091 Win=1040 Len=0 TSval=3191812881 TSecr=1048392310
31	2.046485249	192.168.159.128	10.54.181.105	TCP	1094	55555 → 36360 [PSH, ACK] Seq=2063 Ack=3091 Win=1030 Len=1028 TSval=1048392310 TSecr=3191812880
30	2.045848415	10.54.181.105	192.168.159.128	TCP	60	36360 → 55555 [ACK] Seq=3091 Ack=2063 Win=1024 Len=0 TSval=3191812880 TSecr=1048392310
29	2.045837524	192.168.159.128	10.54.181.105	TCP	60	55555 → 36360 [PSH, ACK] Seq=2061 Ack=3091 Win=1030 Len=2 TSval=1048392310 TSecr=3191812879
28	2.045782961	192.168.159.128	10.54.181.105	TCP	60	55555 → 36360 [ACK] Seq=2061 Ack=3091 Win=1030 Len=0 TSval=1048392309 TSecr=3191812879
27	2.0458067307	10.54.181.105	192.168.159.128	TCP	1094	36360 → 55555 [PSH, ACK] Seq=2063 Ack=2061 Win=1024 Len=1028 TSval=3191812879 TSecr=1048392309
26	2.045816578	192.168.159.128	10.54.181.105	TCP	60	55555 → 36360 [ACK] Seq=2061 Ack=2063 Win=1014 Len=0 TSval=1048392309 TSecr=3191812837
24	2.045821222	10.54.181.105	192.168.159.128	TCP	60	36360 → 55555 [PSH, ACK] Seq=2061 Ack=2061 Win=1024 Len=2 TSval=3191812837 TSecr=1048391308
23	2.045807124	10.124.181.2	10.24.181.2	ICMP	1028	Echo (ping) request id=0x0007, seq=3/768, ttl=64 (reply in 25)
14	1.044755251	10.24.181.2	10.124.181.2	ICMP	1028	Echo (ping) reply id=0x0007, seq=2/512, ttl=64 (request in 13)
22	1.044657924	10.54.181.105	192.168.159.128	TCP	60	36360 → 55555 [ACK] Seq=2061 Ack=2061 Win=1024 Len=0 TSval=3191811879 TSecr=1048391308
21	1.044644842	192.168.159.128	10.54.181.105	TCP	1094	55555 → 36360 [PSH, ACK] Seq=1033 Ack=2061 Win=1014 Len=1028 TSval=1048391308 TSecr=3191811878

Figura 3.8: Captura con wireshark del ping desde RCO-X a 10.24.181.2

Una vez realizado el comando de ping, al comparar el tráfico actual con el registrado durante el ping previamente ejecutado, se observa que en los paquetes ICMP, la dirección IP de origen del Echo Reply ha cambiado y ahora corresponde a la dirección IP de RCO-noX. De manera similar, la dirección IP de destino del Echo Request también ha sido modificada y ahora refleja la dirección IP de RCO-noX. Este cambio en las direcciones IP se debe a las reglas de enrutamiento previamente configuradas en la red.

En particular, se añade una regla en el dispositivo RCO-x con el comando `ip route add 10.24.181.0/24 dev tun3`, que establece que todo el tráfico cuyo destino pertenezca a la red 10.24.181.0/24 será direccionado a través de la interfaz tun3. Esta configuración de enrutamiento provoca que el tráfico con destino a la subred mencionada sea redirigido, lo que genera la modificación observada en las direcciones IP de los paquetes ICMP.

A continuación vamos a modificar las reglas de enrutamiento, tanto en RCO-noX como en RCO-X.

En RCO-noX, se eliminará la ruta hacia la subred 10.54.1.0/24 a través de la interfaz tun0 mediante el comando `ip route del 10.54.181.0/24 dev tun0`. Esta acción implica que cualquier tráfico dirigido a la subred mencionada dejará de ser encaminado por esa interfaz. Posteriormente, se agregará una nueva ruta hacia la misma subred mediante el comando `ip route add 10.54.181.0/24 dev tun0 src 10.24.181.2`. Esta instrucción establecerá que el tráfico hacia la subred 10.54.181.0/24 será enviado a través de la interfaz tun0, pero con dirección IP 10.24.181.2, lo que asegura que los paquetes provenientes de esa subred tendrán esa dirección como origen.

En RCO-X, se realizarán cambios similares. Primero, se eliminará la ruta hacia la subred 10.24.181.0/24 a través de la interfaz tun3 utilizando el comando `ip route del 10.24.181.0/24 dev tun3`. Esto garantiza que el tráfico destinado a esa subred ya no será enviado a través de tun3. A continuación, se añadirá una nueva ruta a la misma subred mediante el comando `ip route add 10.24.181.0/24 dev tun3 src 10.54.181.137`, lo que dirige el tráfico hacia 10.24.181.0/24 a través de la interfaz tun3 con la dirección IP de origen 10.54.181.137.

Una vez realizado los cambios, se realizará desde RCO-X un ping `-s 1000 10.24.181.2`.

```
[root@rco-x simpletun]# ping -s 1000 10.24.181.2
PING 10.24.181.2 (10.24.181.2) 1000(1028) bytes of data.
1008 bytes from 10.24.181.2: icmp_seq=1 ttl=64 time=4.16 ms
1008 bytes from 10.24.181.2: icmp_seq=2 ttl=64 time=43.8 ms
1008 bytes from 10.24.181.2: icmp_seq=3 ttl=64 time=43.7 ms
1008 bytes from 10.24.181.2: icmp_seq=4 ttl=64 time=43.7 ms
^C
--- 10.24.181.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 4.155/33.839/43.780/17.138 ms
[root@rco-x simpletun]#
```

Figura 3.9: ping desde ROC-X a 10.24.181.2 por tunel, reglas cambiadas

Por último, observaremos con el wireshark las direcciones IP fuente y destino.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.54.181.105	10.24.181.2	ICMP	1028	Echo (ping) request id=0x0000, seq=1/256, ttl=64 (reply in 2)
2	0.004121814	10.24.181.2	10.54.181.105	ICMP	1028	Echo (ping) reply id=0x0000, seq=1/256, ttl=64 (request in 1)
3	0.008035614	10.54.181.105	192.168.159.128	TCP	60	36360 → 55555 [PSH, ACK] Seq=1 Ack=1 Win=1056 Len=2 TSval=3192442678 TSecr=1048704837
4	0.002438741	192.168.159.128	10.54.181.105	TCP	60	55555 → 36360 [ACK] Seq=1 Ack=3 Win=1046 Len=0 TSval=1049022112 TSecr=3192442678
5	0.002472695	10.54.181.105	192.168.159.128	TCP	1094	36360 → 55555 [PSH, ACK] Seq=3 Ack=1 Win=1056 Len=1028 TSval=3192442681 TSecr=1049022112
6	0.003212791	192.168.159.128	10.54.181.105	TCP	60	55555 → 36360 [ACK] Seq=1 Ack=1031 Win=1062 Len=0 TSval=1049022112 TSecr=3192442681
7	0.003338660	192.168.159.128	10.54.181.105	TCP	60	55555 → 36360 [PSH, ACK] Seq=1 Ack=1031 Win=1062 Len=2 TSval=1049022112 TSecr=3192442681
8	0.003341622	10.54.181.105	192.168.159.128	TCP	60	36360 → 55555 [ACK] Seq=1031 Ack=3 Win=1056 Len=0 TSval=3192442681 TSecr=1049022112
9	0.004063776	192.168.159.128	10.54.181.105	TCP	1094	55555 → 36360 [PSH, ACK] Seq=3 Ack=1031 Win=1062 Len=1028 TSval=1049022112 TSecr=3192442681
10	0.004074684	10.54.181.105	192.168.159.128	TCP	60	36360 → 55555 [ACK] Seq=1031 Ack=1031 Win=1072 Len=0 TSval=3192442682 TSecr=1049022112
11	1.001512250	10.54.181.105	10.24.181.2	ICMP	1028	Echo (ping) request id=0x0000, seq=2/512, ttl=64 (reply in 13)
12	1.001544547	10.54.181.105	192.168.159.128	TCP	60	36360 → 55555 [PSH, ACK] Seq=1031 Ack=1031 Win=1072 Len=2 TSval=3192443680 TSecr=1049022112
13	1.045269813	10.24.181.2	10.54.181.105	ICMP	1028	Echo (ping) reply id=0x0000, seq=2/512, ttl=64 (request in 11)
14	1.043602448	192.168.159.128	10.54.181.105	TCP	60	55555 → 36360 [ACK] Seq=1031 Ack=1033 Win=1062 Len=0 TSval=1049023152 TSecr=3192443680
15	1.043642598	10.54.181.105	192.168.159.128	TCP	1094	36360 → 55555 [PSH, ACK] Seq=1033 Ack=1031 Win=1072 Len=1028 TSval=3192443722 TSecr=1049023152
16	1.044481772	192.168.159.128	10.54.181.105	TCP	60	55555 → 36360 [ACK] Seq=1031 Ack=2061 Win=1078 Len=0 TSval=1049023153 TSecr=3192443722
17	1.044577151	192.168.159.128	10.54.181.105	TCP	60	55555 → 36360 [PSH, ACK] Seq=1031 Ack=2061 Win=1078 Len=2 TSval=1049023153 TSecr=3192443722
18	1.044590358	10.54.181.105	192.168.159.128	TCP	60	36360 → 55555 [ACK] Seq=2061 Ack=1033 Win=1072 Len=0 TSval=3192443723 TSecr=1049023153
19	1.045208166	192.168.159.128	10.54.181.105	TCP	1094	55555 → 36360 [PSH, ACK] Seq=1033 Ack=2061 Win=1078 Len=1028 TSval=1049023154 TSecr=3192443723
20	1.045219946	10.54.181.105	192.168.159.128	TCP	60	36360 → 55555 [ACK] Seq=2061 Ack=2061 Win=1088 Len=0 TSval=3192443723 TSecr=1049023154
21	2.003406050	10.54.181.105	10.24.181.2	ICMP	1028	Echo (ping) request id=0x0000, seq=3/768, ttl=64 (reply in 23)
22	2.003438603	10.54.181.105	192.168.159.128	TCP	60	36360 → 55555 [PSH, ACK] Seq=2061 Ack=2061 Win=1088 Len=2 TSval=3192444682 TSecr=1049023154
23	2.047094402	10.24.181.2	10.54.181.105	ICMP	1028	Echo (ping) reply id=0x0000, seq=3/768, ttl=64 (request in 21)

Figura 3.10: direcciones con reglas cambiadas

Como era de esperar, debido a las nuevas reglas efectuadas que se han explicado anteriormente, la dirección ip origen de los paquetes ICMP request es 10.54.181.105, que corresponde a la dirección IP de RCO-X y la de destino es 10.24.181.2, que corresponde a la dirección ip de RCO-nox. Algo similar ocurre con el ICMP reply, pero de manera contraria. La dirección ip origen es 10.24.181.2 y la de destino es 10.54.181.105.

3.2 Tarea-2: Túnel como site-to-site

En este segundo apartado vamos a hacer que RCO y RCO-X funcionen como si fueran routers interconectados por un túnel simpletun. Para ello, en primer lugar vamos a activar el forwarding tanto en RCO-X como en RCO-noX, lo que implica que cuando se reciba 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. Esto puede hacerse aplicando el siguiente comando:

```
1 echo 1 | cat >/proc/sys/net/ipv4/ip-forward
```

Si se desea que este cambio sea permanente habrá que editar el fichero sysctl.conf en /etc y descomentar la línea net.ipv4.ip-forward=1. Esta acción se puede realizar con un solo comando:

```
1 # ...
2 # sudo sysctl -w net.ipv4.ip-forward=1
3 # ...
```

A continuación crearemos nuevas reglas para que el tráfico dirigido a la red 10.54.181.0/24 use como Gateway 10.24.181.2.

Para ello crearemos las reglas tanto en el PC anfitrión como en ddwrt-noX para que queden tal como se muestra en las figuras 3.11 y 3.13.

```
root@DD-WRT:~# ip route add 10.54.181.0/24 via 10.24.181.2
root@DD-WRT:~#
```

Figura 3.11: Nueva regla routing ddwrt-noX

```
root@DD-WRT:~# route -n
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
192.168.159.2 0.0.0.0 255.255.255.255 UH 0 0 0 eth0
10.24.181.0 0.0.0.0 255.255.255.0 U 0 0 0 br0
10.54.181.0 10.24.181.2 255.255.255.0 UG 0 0 0 br0
192.168.159.0 0.0.0.0 255.255.255.0 U 0 0 0 eth0
169.254.0.0 0.0.0.0 255.255.0.0 U 0 0 0 br0
127.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 lo
0.0.0.0 192.168.159.2 0.0.0.0 UG 0 0 0 eth0
root@DD-WRT:~#
```

Figura 3.12: route -n ddwrt-noX

```
C:\Windows\system32>route ADD 10.54.181.0 MASK 255.255.255.0 10.24.181.2 METRIC 2 IF 13
Correcto

C:\Windows\system32>route print
=====
Ilista de interfaces
 7...b4 2e 99 af b6 13 .....Realtek Gaming GbE Family Controller
18...c4 73 1e c9 45 52 .....Microsoft Wi-Fi Direct Virtual Adapter
 8...c4 73 1e c9 45 53 .....Microsoft Wi-Fi Direct Virtual Adapter #2
13...00 50 56 c0 00 01 .....VMware Virtual Ethernet Adapter for VMnet1
17...00 50 56 c0 00 08 .....VMware Virtual Ethernet Adapter for VMnet8
14...c4 73 1e c9 45 50 .....802.11n USB Wireless LAN Card
 1.....Software Loopback Interface 1
=====

IPv4 Tabla de enrutamiento
=====
Rutas activas:
Destino de red      Máscara de red      Puerta de enlace    Interfaz  Métrica
0.0.0.0             0.0.0.0             192.168.1.1         192.168.1.65    60
10.24.181.0         255.255.255.0       En vínculo          10.24.181.3     291
10.24.181.3         255.255.255.255     En vínculo          10.24.181.3     291
10.24.181.255       255.255.255.255     En vínculo          10.24.181.3     291
10.54.181.0         255.255.255.0       10.24.181.2         10.24.181.3     37
127.0.0.0           255.0.0.0           En vínculo          127.0.0.1       331
127.0.0.1           255.255.255.255     En vínculo          127.0.0.1       331
127.255.255.255     255.255.255.255     En vínculo          127.0.0.1       331
192.168.1.0         255.255.255.0       En vínculo          192.168.1.65    316
192.168.1.65       255.255.255.255     En vínculo          192.168.1.65    316
```

Figura 3.13: Nueva regla routing anfitrion

También se desea que el tráfico dirigido a 10.24.181.0/24 use como Gateway la IP de ens37 de RCO-X. Para ello se creará las reglas en ddwrt-X para que queden como en la Figura 3.14.

```
root@DD-WRT:~# ip route add 10.24.181.0/24 via 10.54.181.105
root@DD-WRT:~# route -n
Kernel IP routing table
Destination      Gateway          Genmask         Flags Metric Ref    Use Iface
192.168.159.2    0.0.0.0         255.255.255.255 UH      0      0      0 eth0
10.24.181.0      10.54.181.105   255.255.255.0   UG      0      0      0 br0
10.54.181.0      0.0.0.0         255.255.255.0   U        0      0      0 br0
192.168.159.0    0.0.0.0         255.255.255.0   U        0      0      0 eth0
169.254.0.0      0.0.0.0         255.255.0.0     U        0      0      0 br0
127.0.0.0        0.0.0.0         255.0.0.0       U        0      0      0 lo
0.0.0.0          192.168.159.2   0.0.0.0         UG      0      0      0 eth0
root@DD-WRT:~#
```

Figura 3.14: Nueva regla routing ddwrt-X

Vamos a verificar con una serie de órdenes ping que las ips que pertenecen a la red 10.24.181.0/24 comunican con las ips pertenecientes a la red 10.54.181.0/24. Empezamos realizando un ping -s 1000 10.24.181.2


```

root@DD-WRT:~# ping -s 1000 10.24.181.2
PING 10.24.181.2 (10.24.181.2): 1000 data bytes
1008 bytes from 10.24.181.2: seq=0 ttl=63 time=44.122 ms
1008 bytes from 10.24.181.2: seq=1 ttl=63 time=43.597 ms
1008 bytes from 10.24.181.2: seq=2 ttl=63 time=43.533 ms
1008 bytes from 10.24.181.2: seq=3 ttl=63 time=42.740 ms

--- 10.24.181.2 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 42.740/43.498/44.122 ms
root@DD-WRT:~#

```

Figura 3.15: comunican con 10.54, ping ddwrt-X a ddwrt-noX

Como se observa en la Figura 3.15, el ping realizado se ejecuta con éxito. A continuación vamos realizar un ping que comunique ddwrt-X con ddwrt-noX con ping -s 1000 10.54.181.105.

```

root@DD-WRT:~# ping -s 1000 10.54.181.105
PING 10.54.181.105 (10.54.181.105): 1000 data bytes
1008 bytes from 10.54.181.105: seq=0 ttl=63 time=2.947 ms
1008 bytes from 10.54.181.105: seq=1 ttl=63 time=44.611 ms
1008 bytes from 10.54.181.105: seq=2 ttl=63 time=43.473 ms
1008 bytes from 10.54.181.105: seq=3 ttl=63 time=43.942 ms
1008 bytes from 10.54.181.105: seq=4 ttl=63 time=43.769 ms
1008 bytes from 10.54.181.105: seq=5 ttl=63 time=43.530 ms

--- 10.54.181.105 ping statistics ---
6 packets transmitted, 6 packets received, 0% packet loss
round-trip min/avg/max = 2.947/37.045/44.611 ms
root@DD-WRT:~#

```

Figura 3.16: Enter Caption

Y realizamos una captura con wireshark en RCO-X.

No.	Time	Source	Destination	Protocol	Length	Info
3	0.000040925	10.54.181.101	10.24.181.2	ICMP	1042	Echo (ping) request id=0x2d09, seq=0/0, ttl=64 (reply in 12)
4	0.000041199	10.54.181.105	192.168.159.128	TCP	60	6636188 → 55555 [PSH, ACK] Seq=1 Ack=1 Win=437 Len=2 TSval=216478515 TSecr=120970553
5	0.000938023	192.168.159.128	10.54.181.105	TCP	66	55555 → 36188 [ACK] Seq=1 Ack=3 Win=436 Len=0 TSval=1209707552 TSecr=216478515
6	0.000954721	10.54.181.105	192.168.159.128	TCP	1094	36188 → 55555 [PSH, ACK] Seq=3 Ack=1 Win=437 Len=1028 TSval=216478515 TSecr=1209707552
7	0.001062140	192.168.159.128	10.54.181.105	TCP	66	55555 → 36188 [ACK] Seq=1 Ack=1031 Win=452 Len=0 TSval=1209707553 TSecr=216478515
8	0.001073946	192.168.159.128	10.54.181.105	TCP	66	55555 → 36188 [PSH, ACK] Seq=1 Ack=1031 Win=452 Len=2 TSval=1209707553 TSecr=216478515
9	0.001079556	10.54.181.105	192.168.159.128	TCP	66	36188 → 55555 [ACK] Seq=1031 Ack=3 Win=437 Len=0 TSval=216478516 TSecr=1209707553
10	0.002443405	192.168.159.128	10.54.181.105	TCP	1094	55555 → 36188 [PSH, ACK] Seq=3 Ack=1031 Win=452 Len=1028 TSval=1209707553 TSecr=216478516
11	0.002454360	10.54.181.105	192.168.159.128	TCP	66	36188 → 55555 [ACK] Seq=1031 Ack=1031 Win=453 Len=0 TSval=216478517 TSecr=1209707553
2	0.002502538	10.24.181.2	10.54.181.101	ICMP	1028	Echo (ping) reply id=0x2d09, seq=0/0, ttl=64 (request in 1)
12	0.002512311	10.24.181.2	10.54.181.101	ICMP	1042	Echo (ping) reply id=0x2d09, seq=0/0, ttl=63 (request in 3)
13	0.002545551	10.54.181.101	10.24.181.2	ICMP	1042	Echo (ping) request id=0x2d09, seq=1/256, ttl=64 (reply in 24)
13	0.002082529	10.54.181.101	10.24.181.2	ICMP	1028	Echo (ping) request id=0x2d09, seq=1/256, ttl=63 (reply in 14)
16	0.002124746	10.54.181.105	192.168.159.128	TCP	66	36188 → 55555 [PSH, ACK] Seq=1031 Ack=1031 Win=453 Len=2 TSval=216479517 TSecr=1209707553
17	0.043846112	192.168.159.128	10.54.181.105	TCP	66	55555 → 36188 [ACK] Seq=1031 Ack=1033 Win=452 Len=0 TSval=1209708595 TSecr=216479517
18	0.043884032	10.54.181.105	192.168.159.128	TCP	1094	36188 → 55555 [PSH, ACK] Seq=1033 Ack=1031 Win=453 Len=1028 TSval=216479558 TSecr=1209708595
19	0.044097002	192.168.159.128	10.54.181.105	TCP	66	55555 → 36188 [ACK] Seq=1031 Ack=2061 Win=408 Len=0 TSval=1209708596 TSecr=216479558
20	0.044710616	192.168.159.128	10.54.181.105	TCP	66	55555 → 36188 [PSH, ACK] Seq=1031 Ack=2061 Win=408 Len=2 TSval=1209708596 TSecr=216479558
21	0.044716741	10.54.181.105	192.168.159.128	TCP	66	36188 → 55555 [ACK] Seq=2061 Ack=1033 Win=453 Len=0 TSval=216479559 TSecr=1209708596
22	0.045386199	192.168.159.128	10.54.181.105	TCP	1094	55555 → 36188 [PSH, ACK] Seq=1033 Ack=2061 Win=408 Len=1028 TSval=1209708596 TSecr=216479559
23	0.045397658	10.54.181.105	192.168.159.128	TCP	66	36188 → 55555 [ACK] Seq=2061 Ack=2061 Win=470 Len=0 TSval=216479560 TSecr=1209708596
14	0.045444545	10.24.181.2	10.54.181.101	ICMP	1028	Echo (ping) reply id=0x2d09, seq=1/256, ttl=64 (request in 13)
24	0.045450249	10.24.181.2	10.54.181.101	ICMP	1042	Echo (ping) reply id=0x2d09, seq=1/256, ttl=63 (request in 15)

Frame 1: 1028 bytes on wire (8224 bits), 1028 bytes captured (8224 bits) on interface 0
 Interface id: 0 (tun3)
 Encapsulation type: Linux
 Arrival Time: Nov 29, 2024 09:39:59.703138016 CET
 [Time shift for this packet: 0.000000000 seconds]
 Epoch Time: 1732837199.703138016 seconds
 [Time delta from previous captured frame: 0.000000000 seconds]
 [Time delta from previous displayed frame: 0.000000000 seconds]
 0000 45 00 04 04 00 00 00 00 3f 01 b9 43 0a 36 b5 65 E.....@.7..C.6.e
 0010 0a 18 05 02 00 00 00 00 2d 09 00 00 99 c5 58 70Xp
 0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Figura 3.17: Wireshark ping ddwrt-noX a ddwrt-X

En la captura de tráfico realizada con Wireshark (ver Figura 3.17), se puede observar, por línea de salida del comando ping y en orden cronológico, dos mensajes ICMP Echo

Request, ocho paquetes TCP y dos mensajes ICMP Echo Reply. La presencia de dos solicitudes (Echo Request) y dos respuestas (Echo Reply) se debe a que se han capturado los paquetes a través de las dos interfaces de red (tun3 y ens 37), lo que provoca que los mismos paquetes ping se muestren duplicados en la captura.

También se observa que el primer mensaje Echo Request presenta una cabecera Ethernet, mientras que el segundo Echo Request no la tiene (ver Figuras 3.18 y 3.19). Esta diferencia se puede explicar por las características de las interfaces involucradas. Sabemos que en la interfaz tun3 no existe hardware físico, ya que es una interfaz emulada por software, por lo tanto, no utiliza el protocolo Ethernet. En consecuencia, el Echo Request sin cabecera Ethernet es el que transita a través de esta interfaz tun3. En cambio, el Echo Request que incluye la cabecera Ethernet corresponde a una interfaz física (ens37), que requiere el protocolo Ethernet para la transmisión de los paquetes, lo que indica que este último es el que viaja por la interfaz ens37.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	fe80::250:56ff:fe38...	ff02::2	ICMPv6	70	Router Solicitation from 00:50:56:38:60:9c
4	2.033957744	10.54.181.101	10.24.181.2	ICMP	1042	Echo (ping) request id=0x3a09, seq=0/0, ttl=64 (reply in 13)
2	2.033959587	10.54.181.101	10.24.181.2	ICMP	1028	Echo (ping) request id=0x3a09, seq=0/0, ttl=63 (reply in 3)
5	2.034039835	10.54.181.105	192.168.159.128	TCP	68	36188 → 55555 [PSH, ACK] Seq=1 Ack=1 Win=614 Len=2 TSval=216597006 TS
6	2.034932488	192.168.159.128	10.54.181.105	TCP	66	55555 → 36188 [ACK] Seq=1 Ack=3 Win=612 Len=0 TSval=1209826043 TSecr=
7	2.034961406	10.54.181.105	192.168.159.128	TCP	1094	36188 → 55555 [PSH, ACK] Seq=3 Ack=1 Win=614 Len=1028 TSval=216597007
8	2.035687910	192.168.159.128	10.54.181.105	TCP	66	55555 → 36188 [ACK] Seq=1 Ack=1031 Win=628 Len=0 TSval=1209826044 TSe
9	2.035697638	192.168.159.128	10.54.181.105	TCP	68	55555 → 36188 [PSH, ACK] Seq=1 Ack=1031 Win=628 Len=2 TSval=120982604
10	2.076224780	10.54.181.105	192.168.159.128	TCP	66	36188 → 55555 [ACK] Seq=1031 Ack=3 Win=614 Len=0 TSval=216597048 TSec
11	2.077068739	192.168.159.128	10.54.181.105	TCP	1094	55555 → 36188 [PSH, ACK] Seq=3 Ack=1031 Win=628 Len=1028 TSval=120982
12	2.077086661	10.54.181.105	192.168.159.128	TCP	66	36188 → 55555 [ACK] Seq=1031 Ack=1031 Win=630 Len=0 TSval=216597049 T
3	2.077143863	10.24.181.2	10.54.181.101	ICMP	1028	Echo (ping) reply id=0x3a09, seq=0/0, ttl=64 (request in 2)
13	2.077155740	10.24.181.2	10.54.181.101	ICMP	1042	Echo (ping) reply id=0x3a09, seq=0/0, ttl=63 (request in 4)
16	3.036601867	10.54.181.101	10.24.181.2	ICMP	1042	Echo (ping) request id=0x3a09, seq=1/256, ttl=64 (reply in 25)
14	3.036629188	10.54.181.101	10.24.181.2	ICMP	1028	Echo (ping) request id=0x3a09, seq=1/256, ttl=63 (reply in 15)

Frame 4: 1042 bytes on wire (8336 bits), 1042 bytes captured (8336 bits) on interface 1
 Ethernet II, Src: Vmware_3e:ab:c9 (00:50:56:3e:ab:c9), Dst: Vmware_38:60:9c (00:50:56:38:60:9c)
 Internet Protocol Version 4, Src: 10.54.181.101, Dst: 10.24.181.2
 Internet Control Message Protocol

Figura 3.18: ICMP cabecera Ethernet

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	fe80::250:56ff:fe38...	ff02::2	ICMPv6	70	Router Solicitation from 00:50:56:38:60:9c
4	2.033957744	10.54.181.101	10.24.181.2	ICMP	1042	Echo (ping) request id=0x3a09, seq=0/0, ttl=64 (reply in 13)
2	2.033959587	10.54.181.101	10.24.181.2	ICMP	1028	Echo (ping) request id=0x3a09, seq=0/0, ttl=63 (reply in 3)
5	2.034039835	10.54.181.105	192.168.159.128	TCP	68	36188 → 55555 [PSH, ACK] Seq=1 Ack=1 Win=614 Len=2 TSval=216597006 TS
6	2.034932488	192.168.159.128	10.54.181.105	TCP	66	55555 → 36188 [ACK] Seq=1 Ack=3 Win=612 Len=0 TSval=1209826043 TSecr=
7	2.034961406	10.54.181.105	192.168.159.128	TCP	1094	36188 → 55555 [PSH, ACK] Seq=3 Ack=1 Win=614 Len=1028 TSval=216597007
8	2.035687910	192.168.159.128	10.54.181.105	TCP	66	55555 → 36188 [ACK] Seq=1 Ack=1031 Win=628 Len=0 TSval=1209826044 T
9	2.035697638	192.168.159.128	10.54.181.105	TCP	68	55555 → 36188 [PSH, ACK] Seq=1 Ack=1031 Win=628 Len=2 TSval=120982604
10	2.076224780	10.54.181.105	192.168.159.128	TCP	66	36188 → 55555 [ACK] Seq=1031 Ack=3 Win=614 Len=0 TSval=216597048 TS
11	2.077068739	192.168.159.128	10.54.181.105	TCP	1094	55555 → 36188 [PSH, ACK] Seq=3 Ack=1031 Win=628 Len=1028 TSval=120982
12	2.077086661	10.54.181.105	192.168.159.128	TCP	66	36188 → 55555 [ACK] Seq=1031 Ack=1031 Win=630 Len=0 TSval=216597049 T
3	2.077143863	10.24.181.2	10.54.181.101	ICMP	1028	Echo (ping) reply id=0x3a09, seq=0/0, ttl=64 (request in 2)
13	2.077155740	10.24.181.2	10.54.181.101	ICMP	1042	Echo (ping) reply id=0x3a09, seq=0/0, ttl=63 (request in 4)
16	3.036601867	10.54.181.101	10.24.181.2	ICMP	1042	Echo (ping) request id=0x3a09, seq=1/256, ttl=64 (reply in 25)
14	3.036629188	10.54.181.101	10.24.181.2	ICMP	1028	Echo (ping) request id=0x3a09, seq=1/256, ttl=63 (reply in 15)

Frame 2: 1028 bytes on wire (8224 bits), 1028 bytes captured (8224 bits) on interface 0
 Raw packet data
 Internet Protocol Version 4, Src: 10.54.181.101, Dst: 10.24.181.2
 Internet Control Message Protocol

Figura 3.19: ICMP sin cabecera Ethernet

CAPÍTULO 4

Modificación de simpletun y funcionamiento del programa modificado

A continuación, realizaremos modificaciones en el código simpletun.c para incluir un cifrado en el túnel.

Este sistema se dividirá en dos etapas: primero implementaremos un cifrado sencillo conocido como Caesar y, seguidamente, aplicaremos un método ligeramente más avanzado basado en el operador XOR.

4.1 Tarea-3: Cifrado Caesar

El cifrado Caesar es un tipo de cifrado por sustitución, donde cada letra del texto original (texto plano) se reemplaza por otra que se encuentra un número fijo de posiciones adelante en el alfabeto. Este número fijo es la clave del cifrado.

Por ejemplo, un Caesar-3 codifica la "B" como "E", la "C" como "F", etc. (ver Figura 4.1)

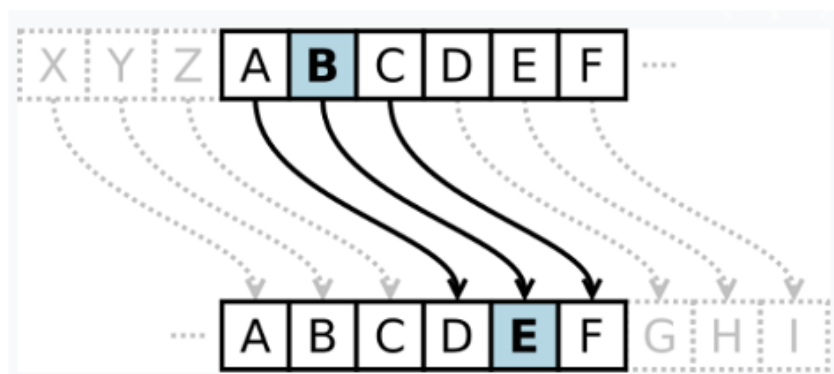


Figura 4.1: Ejemplo de cifrado Caesar

Implementamos un Caesar-N, donde N es el número de nuestro grupo asignado en prácticas (181).

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

1. Codificación Caesar-181: $c = (a + 181) \% 256$
2. Decodificación Caesar-181: $a = (c + 256 - 181) \% 256$

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

Es importante mencionar que, aunque utilizamos caracteres como ejemplo, los datos que se transmiten y reciben son en formato binario, por lo que el cifrado se aplica a todos los bytes de la información.

4.1.1. Implementación

Procedemos a modificar el código de simpletun.c en la máquina RCO-X y RCO-noX. Primero definimos unas variables (X e Y) para así poder itear en los bucles.

```

320  /* use select() to handle two descriptors at once */
321  maxfd = (tap_fd > net_fd)?tap_fd:net_fd;
322
323  while(1) {
324      int ret;
325      int x;
326      int y;
327      fd_set rd_set;

```

Figura 4.2: Variables añadidas

Una vez definidas las variables X e Y, podremos utilizarlas en los bucles necesarios para modificar los datos según el desplazamiento determinado por el cifrado Caesar-181, lo que nos permitirá cifrar y descifrar la información.

Para el cifrado, desplazamos cada byte 181 posiciones hacia adelante. Es decir, reemplazamos $buffer[x]$ por $buffer[x] + 181$ y se le aplica el módulo 256 (ver Figura 4.3).

```

343  if(FD_ISSET(tap_fd, &rd_set)){
344      /* data from tun/tap: just read it and write it to the network */
345
346      nread = cread(tap_fd, buffer, BUFSIZE);
347
348      tap2net++;
349      do_debug("TAP2NET %lu: Read %d bytes from the tap interface\n", tap2net, nread);
350
351      /* Codificación Caesar-N: c = (a + N) % 256 */
352      for(x=0; x<nread; x++){
353          buffer[x] = (buffer[x]+181)%256
354      }
355
356      /* write length + packet */
357      plength = htons(nread);
358      nwrite = cwrite(net_fd, (char *)&plength, sizeof(plength));
359      nwrite = cwrite(net_fd, buffer, nread);
360
361      do_debug("TAP2NET %lu: Written %d bytes to the network\n", tap2net, nwrite);
362  }

```

Figura 4.3: Codificación Caesar

Para descifrar, necesitamos revertir el desplazamiento de 181 posiciones. Sin embargo, si el valor resultante es menor que 0, se podrían generar valores negativos, lo cual no es permitido. Para solucionar esto, sumamos 256 antes de restar los 181. Así, en lugar de

hacer `buffer[y] - 181`, hacemos `buffer[y] + 256 - 181` y se le aplica seguidamente el módulo 256 (ver Figura 4.4).

```

364 if(FD_ISSET(net_fd, &rd_set)){
365     /* data from the network: read it, and write it to the tun/tap interface.
366      * We need to read the length first, and then the packet */
367
368     /* Read length */
369     nread = read_n(net_fd, (char *)&plength, sizeof(plength));
370     if(nread == 0) {
371         /* ctrl-c at the other end */
372         break;
373     }
374
375     net2tap++;
376
377     /* read packet */
378     nread = read_n(net_fd, buffer, ntohs(plength));
379     do_debug("NET2TAP %lu: Read %d bytes from the network\n", net2tap, nread);
380
381     /* now buffer[] contains a full packet or frame, write it into the tun/tap interface */
382     /* Decodificación Caesar-N: a = (c + 256 - N) % 256 */
383     for(y=0; y<nread; y++){
384         buffer[y] = (buffer[y]+256-181)%256
385     }
386     nwrite = cwrite(tap_fd, buffer, nread);
387     do_debug("NET2TAP %lu: Written %d bytes to the tap interface\n", net2tap, nwrite);
388 }
389
390

```

Figura 4.4: Descodificación Caesar

Se le aplica el módulo 256, tanto en el cifrado como en el descifrado, para evitar un posible desbordamiento y asegurando así que el búfer se comporte de manera circular.

4.1.2. Comprobación

Para verificar que el código funciona correctamente, realizaremos un ping desde `ddwrt-noX` hacia `ddwrt-x` utilizando las direcciones IP de sus respectivas redes VMNet. El resultado de este ping se puede ver en la Figura 4.5.

```

root@DD-WRT:~# ping -s 1000 10.54.181.101
PING 10.54.181.101 (10.54.181.101): 1000 data bytes
1008 bytes from 10.54.181.101: seq=0 ttl=62 time=3.251 ms
1008 bytes from 10.54.181.101: seq=1 ttl=62 time=44.093 ms
1008 bytes from 10.54.181.101: seq=2 ttl=62 time=43.677 ms
1008 bytes from 10.54.181.101: seq=3 ttl=62 time=44.113 ms
1008 bytes from 10.54.181.101: seq=4 ttl=62 time=43.760 ms

--- 10.54.181.101 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 3.251/35.778/44.113 ms
root@DD-WRT:~#

```

Figura 4.5: Ping para ver el funcionamiento

Una vez verificado el funcionamiento, capturaremos el mismo ping con Wireshark desde RCO-X.

Figura 4.6: Ping con cifrado Caesar(I)

No.	Time	Source	Titation	Protocol	Length	Info
12	8.827777381	192.168.159.128	154.181.105	TCP	68 51766	- 51766 [PSH, ACK] Seq=51 Ack=1 Win=227 Len=2 TSval=1359716714 TSecr=4102884046
13	8.827890143	154.181.105	192.168.159.128	TCP	66 51766	- 55555 [ACK] Seq=1 Ack=53 Win=229 Len=0 TSval=4102892874 TSecr=1359716714
14	8.828676472	192.168.159.128	154.181.105	TCP	1094 55555	- 51766 [PSH, ACK] Seq=53 Ack=1 Win=227 Len=1028 TSval=1359716715 TSecr=4102892874
15	8.828686429	154.181.105	192.168.159.128	TCP	66 51766	- 55555 [ACK] Seq=1 Ack=1081 Min=245 Len=0 TSval=4102892875 TSecr=1359716715
16	8.828784777	154.181.105	192.168.159.128	ICMP	1028 Echo (ping) request	id=0xe408, seq=0/, ttl=63 (reply in 11)
17	8.828855443	154.181.105	154.181.105	ICMP	1042 Echo (ping) request	id=0xe408, seq=0/, ttl=62 (reply in 17)
17	8.829217383	154.181.105	154.181.105	ICMP	1042 Echo (ping) reply	id=0xe408, seq=0/, ttl=64 (request in 16)
11	8.829228855	154.181.105	154.181.105	ICMP	1028 Echo (ping) reply	id=0xe408, seq=0/, ttl=63 (request in 10)
18	8.829386127	154.181.105	192.168.159.128	TCP	68 51766	- 55555 [PSH, ACK] Seq=1 Ack=1081 Min=245 Len=2 TSval=4102892875 TSecr=1359716715
19	8.830093606	192.168.159.128	154.181.105	TCP	66 55555	- 51766 [ACK] Seq=1081 Ack=3 Win=227 Len=0 TSval=1359716716 TSecr=4102892875
20	8.83016268	154.181.105	192.168.159.128	TCP	1094 51766	- 55555 [PSH, ACK] Seq=3 Ack=1081 Min=245 Len=1028 TSval=4102892876 TSecr=1359716716
21	8.830811521	192.168.159.128	154.181.105	TCP	66 55555	- 51766 [ACK] Seq=1081 Ack=1031 Win=243 Len=0 TSval=1359716717 TSecr=4102892876
22	8.832198678	192.168.159.128	154.181.105	TCP	68 55555	- 51766 [ACK] Seq=1081 Ack=1031 Win=243 Len=2 TSval=1359717714 TSecr=4102892876
25	9.868618272	154.181.105	192.168.159.128	TCP	67 51766	- 51766 [ACK] Seq=1083 Ack=1083 Win=245 Len=0 TSval=1359717715 TSecr=1359717715
26	9.869690333	192.168.159.128	154.181.105	TCP	1094 55555	- 51766 [PSH, ACK] Seq=1083 Ack=1031 Win=243 Len=1028 TSval=1359717756 TSecr=4102893915
27	9.869629970	154.181.105	192.168.159.128	TCP	66 51766	- 55555 [ACK] Seq=1031 Ack=2111 Win=261 Len=0 TSval=4102893916 TSecr=1359717756
28	9.869790918	154.181.105	192.168.159.128	ICMP	1028 Echo (ping) request	id=0xe408, seq=1/256, ttl=63 (reply in 23)
<hr/>						
Internet Protocol Version 4, Src: 192.168.159.128, Dst: 154.181.105						
Transmission Control Protocol, Src Port: 55555, Dst Port: 51766, Seq: 53, Ack: 1, Len: 1028						
Data (1028 bytes)						
Data: fab5b9bb5bf5bf5f4d6eeffbf6cdeabbbf6beba1abb5daace...						
[Length: 1028]						
<hr/>						
0040	2d 4e f5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5	f4 ce f5 f4 b5 b5 ff df c0				
0060	af ab 0a 1b b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5	b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5				
0080	ad e7 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5	b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5				
00a0	b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5	b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5				
00c0	b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5	b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5				
00e0	b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5	b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5				
0100	b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5	b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5				
0120	b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5	b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5				
0140	b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5	b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5 b5				

Figura 4.7: Ping con cifrado Caesar(II)

La técnica de cifrado utilizada se fundamenta en la operación XOR, es un método común en la criptografía para proteger la información transmitida. En este caso, la clave secreta usada para realizar el XOR es una parte del archivo binario del comando 'ls', el cual se encuentra en la máquina cliente y servidor. La clave y el mensaje se combinan byte a byte usando la operación XOR en bloques de tamaño fijo, igual al tamaño del buffer. Este proceso garantiza que el texto cifrado sea completamente reversible, ya que aplicar nuevamente XOR con la misma clave sobre el texto cifrado recupera el mensaje original.

Este enfoque proporciona un nivel de seguridad adicional al usar como clave un archivo binario estándar del sistema operativo.

4.2.1. Implementación

En este fragmento del código se establecen las variables requeridas para hacer el cifrado avanzado. El puntero FILE *key apunta a un archivo que se usará como clave para el cifrado. La variable size_t bytes_read guarda la cantidad de bytes leídos del archivo clave, facilitando el manejo de la sección que se utilizará.

Por último, bufcif[BUFSIZE] es un conjunto de datos utilizado como almacenamiento temporal de la información extraída del archivo, que será empleado como clave en la operación XOR.

```
1 /* Variables para cifrado avanzado */
2 FILE *key;
3 size_t bytes_read;
4 char buf_cif[BUFSIZE];
```

El bloque de código que inicia el buffer comienza abriendo el archivo binario '/bin/ls' en modo lectura. Se comprueba si la acción tuvo éxito; si no, se produce un error con el mensaje adecuado y se detiene el proceso.

Después, se leen la cantidad de bytes del archivo igual al tamaño del buffer bufcif, definido por BUFSIZE. Los datos leídos del archivo se convierten en la clave que se usará para el cifrado y descifrado de los datos en tránsito. Finalmente, el archivo se cierra para liberar los recursos.

```
1 /* Inicializacion del buffer */
2 #key = fopen("/bin/ls", "r");
3 #
4 #if (key == NULL) {
5 #    perror("Error abriendo el archivo");
6 #    return 1; // Salir si hubo un error
7 #}
8 #
9 #bytes_read = fread(buf_cif, 1, sizeof(buf_cif), key);
10 #
11 #fclose(key);
```

Seguidamente, definimos las variables xs y ys. Estos dos enteros son contadores empleados para recorrer los datos durante el cifrado y descifrado. El objetivo es examinar los bytes de los paquetes guardados en el buffer principal y realizar la operación XOR con los bytes correspondientes de la clave almacenada en bufcif.

```
1 #while(1) {
2 # ...
3 # int x_s;
4 # int y_s;
5 # ...
```

Para hacer el cifrado 'secreto', se utiliza un bucle for para recorrer los bytes del paquete leído desde la interfaz de red o tap. Cada byte en el buffer principal (buffer[xs]) se modifica aplicándole la operación XOR con el byte correspondiente de la clave almacenada en bufcif[xs].

Este método modifica los datos cifrados haciendo que sea difícil de interpretar la información sin conocer la clave exacta. Al final del proceso, el buffer modificado se envía a través de la red.

```

1 /* Cifrado secreto XOR */
2 #for(x_s=0; x_s<nread; x_s++){
3 #   buffer[x_s] = (buffer[x_s]^buf_cif[x_s]);
4 #}

```

Similar al proceso de cifrado, en el descifrado se utiliza un bucle for para recorrer los datos que se reciben desde la red y están almacenados en el buffer principal (buffer[ys]). Se aplica e nuevo la operación XOR con el byte correspondiente de la clave (bufcif[ys]).

Gracias a la propiedad conmutativa y reversible del operador XOR, este paso restaura los datos originales. Esto significa que los datos cifrados se transforman nuevamente en el formato original del paquete antes de ser escritos en la interfaz tap.

```

1 /* Descifrado secreto XOR */
2 #   for(y_s=0; y_s<nread; y_s++){
3 #       buffer[y_s] = (buffer[y_s]^buf_cif[y_s]);
4 #}

```

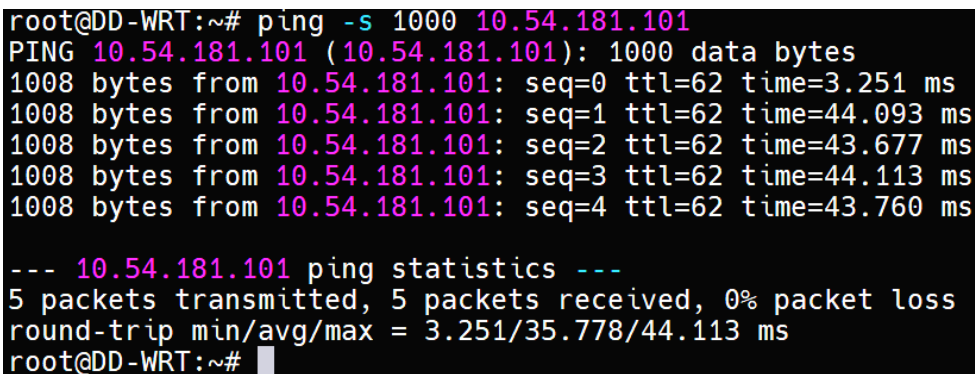
Una vez disponemos del mismo 'simpletun.c' en la máquina cliente y servidor, eliminamos el archivo simpletun compilado anteriormente y compilamos el nuevo:

```

1 gcc simpletun.c -o simpletun

```

Ahora, matamos el proceso simpletun anterior y ejecutamos el nuevo simpletun con cifrado secreto. Vamos a comprobar el correcto funcionamiento del cifrado secreto, para ello realizaremos una prueba de ping desde ddwrt-noX a ddwrt-X. como se puede ver en la Figura 4.8.



```

root@DD-WRT:~# ping -s 1000 10.54.181.101
PING 10.54.181.101 (10.54.181.101): 1000 data bytes
1008 bytes from 10.54.181.101: seq=0 ttl=62 time=3.251 ms
1008 bytes from 10.54.181.101: seq=1 ttl=62 time=44.093 ms
1008 bytes from 10.54.181.101: seq=2 ttl=62 time=43.677 ms
1008 bytes from 10.54.181.101: seq=3 ttl=62 time=44.113 ms
1008 bytes from 10.54.181.101: seq=4 ttl=62 time=43.760 ms

--- 10.54.181.101 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 3.251/35.778/44.113 ms
root@DD-WRT:~#

```

Figura 4.8: ping cifrado secreto

4.2.2. Comprobación

En las imágenes capturadas en Wireshark, se observa el funcionamiento correcto del cifrado XOR comparando las tramas del mensaje sin cifrar ICMP request y la del mensaje cifrado TCP de datos. La diferencia entre estas tramas es clave para demostrar el cifrado de los datos originales.

En la primera Figura 4.9, correspondiente al mensaje ICMP request, los datos enviados no están cifrados.

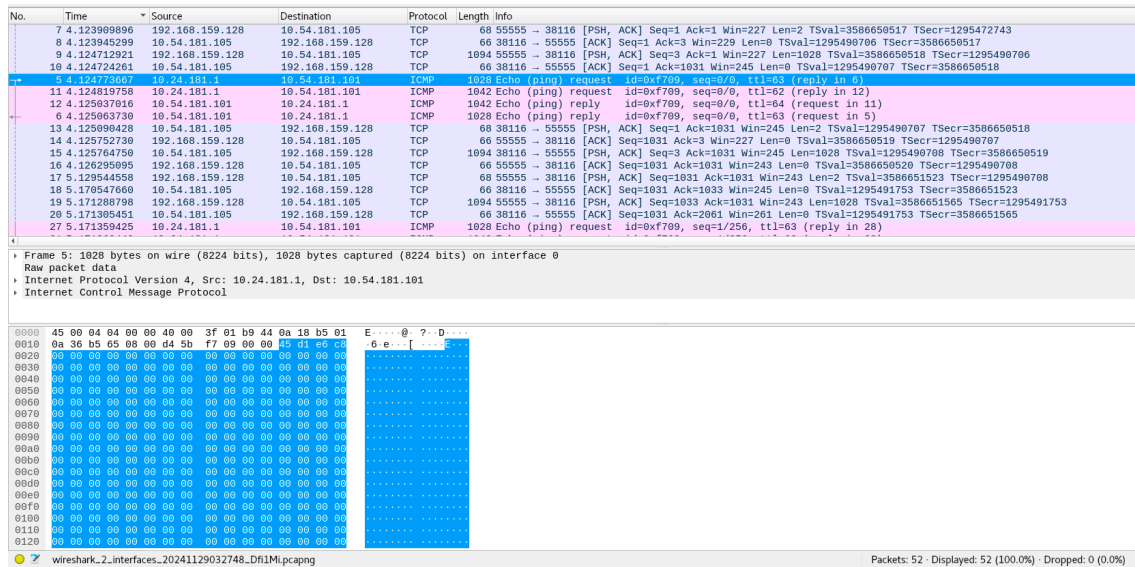


Figura 4.9: Mensaje sin cifrado secreto

La segunda 4.10, correspondiente a la trama TCP, se observa el mensaje después de ser cifrado con XOR utilizando la clave generada a partir del archivo binario /bin/lis.

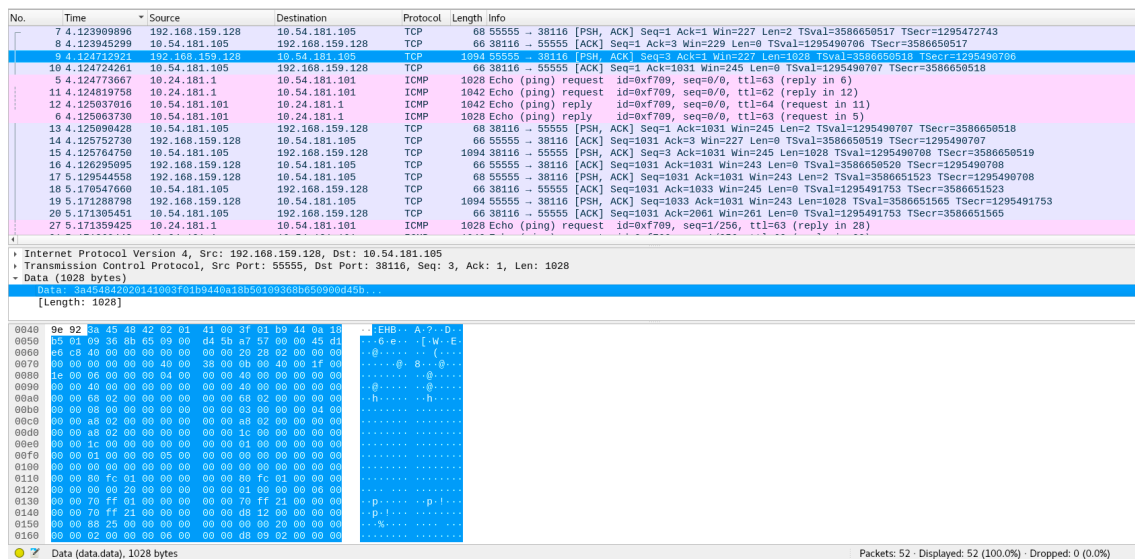


Figura 4.10: mensaje con cifrado secreto

Podemos verificar que el cifrado está funcionando correctamente y protege los datos enviados.

CAPÍTULO 5

Conclusiones

En el presente trabajo, hemos explorado el proceso de configuración de una VPN utilizando `simpletun` y el funcionamiento de las interfaces Tun/Tap, mediante las cuales hemos logrado conectar dos máquinas situadas en diferentes redes a través de un túnel virtual.

Hemos realizado modificaciones en las reglas de enrutamiento mediante el comando `ip route add`, llevado a cabo pruebas de conectividad utilizando el comando `ping`, y empleado la herramienta Wireshark para analizar con mayor detalle los paquetes transmitidos a través de nuestra red virtual, lo que nos a permitido obtener una comprensión más profunda del funcionamiento interno de las interfaces Tun/Tap en el entorno de nuestra red virtual. Adicionalmente, se ha realizado una modificación del código fuente de `simpletun`, específicamente el archivo `simpletun.c`, para integrar métodos de cifrado simples en el túnel, tales como los cifrados Caesar y XOR. Esta implementación nos ha permitido brindar una mayor protección a los datos transmitidos a través del túnel y acercarnos a los principios fundamentales de una VPN.

En resumen, `simpletun` se presenta como una herramienta robusta y accesible para la implementación de VPNs, destacándose por su facilidad de configuración, su capacidad para crear túneles TCP y su flexibilidad en su aplicación en redes virtuales. La combinación de su simplicidad operativa con mejoras en seguridad, como el cifrado, lo convierte en una opción eficiente para la creación de redes privadas virtuales.

Bibliografía

- [1] William Buchanan. *Cryptography*. River Publishers, 1st edition, 2017.
- [2] Al Sweigart. *Cracking Codes with Python: An Introduction to Building and Breaking Ciphers*. No Starch Press, 1st edition, 2018.
- [3] Eric F Crist and Jan Just Keijser. *Mastering OpenVPN: Master Building and Integrating Secure Private Networks Using OpenVPN*. Packt Publishing, 1st edition, 2015.
- [4] Joshua Feldman Eric Conrad. *Cryptography. Eleventh Hour CISSP (Second Edition)*, 11(5):77–96, 2014. Disponible en <https://www.sciencedirect.com/topics/computer-science/symmetric-encryption#:~:text=Symmetric%20encryption%20uses%20one%20key,kept%20secret%20from%20third%20parties>.
- [5] geeksforgeeks. Caesar cipher in cryptography, 2024. Disponible en <https://www.geeksforgeeks.org/caesar-cipher-in-cryptography/>, consultada el 25/11/2024].

APÉNDICE A

Listados

Aquí hemos puesto el código del `simpletun.c` con el cifrado secreto y el cifrado Caesar comentado. A.1

```
1  /*****
2  * simpletun.c
3  *
4  * A simplistic , simple-minded, naive tunnelling program using tun/tap
5  * interfaces and TCP. Handles (badly) IPv4 for tun, ARP and IPv4 for
6  * tap. DO NOT USE THIS PROGRAM FOR SERIOUS PURPOSES.
7  *
8  * You have been warned.
9  *
10 * (C) 2009 Davide Brini.
11 *
12 * DISCLAIMER AND WARNING: this is all work in progress. The code is
13 * ugly, the algorithms are naive, error checking and input validation
14 * are very basic, and of course there can be bugs. If that's not enough,
15 * the program has not been thoroughly tested, so it might even fail at
16 * the few simple things it should be supposed to do right.
17 * Needless to say, I take no responsibility whatsoever for what the
18 * program might do. The program has been written mostly for learning
19 * purposes, and can be used in the hope that is useful, but everything
20 * is to be taken "as is" and without any kind of warranty, implicit or
21 * explicit. See the file LICENSE for further details.
22 *****/
23
24 #include <stdio.h>
25 #include <stdlib.h>
26 #include <string.h>
27 #include <unistd.h>
28 #include <sys/socket.h>
29 #include <linux/if.h>
30 #include <linux/if_tun.h>
31 #include <sys/types.h>
32 #include <sys/ioctl.h>
33 #include <sys/stat.h>
34 #include <fcntl.h>
35 #include <arpa/inet.h>
36 #include <sys/select.h>
37 #include <sys/time.h>
38 #include <errno.h>
39 #include <stdarg.h>
40
41 /* buffer for reading from tun/tap interface , must be >= 1500 */
42 #define BUFSIZE 2000
43 #define CLIENT 0
44 #define SERVER 1
```

```

45 #define PORT 55555
46
47 /* some common lengths */
48 #define IP_HDR_LEN 20
49 #define ETH_HDR_LEN 14
50 #define ARP_PKT_LEN 28
51
52 int debug;
53 char *progname;
54
55 /*****
56  * tun_alloc: allocates or reconnects to a tun/tap device. The caller
57  *             needs to reserve enough space in *dev.
58  *****/
59 int tun_alloc(char *dev, int flags) {
60
61     struct ifreq ifr;
62     int fd, err;
63
64     if( (fd = open("/dev/net/tun", O_RDWR)) < 0 ) {
65         perror("Opening /dev/net/tun");
66         return fd;
67     }
68
69     memset(&ifr, 0, sizeof(ifr));
70
71     ifr.ifr_flags = flags;
72
73     if (*dev) {
74         strncpy(ifr.ifr_name, dev, IFNAMSIZ);
75     }
76
77     if( (err = ioctl(fd, TUNSETIFF, (void *)&ifr)) < 0 ) {
78         perror("ioctl(TUNSETIFF)");
79         close(fd);
80         return err;
81     }
82
83     strcpy(dev, ifr.ifr_name);
84
85     return fd;
86 }
87
88 /*****
89  * cread: read routine that checks for errors and exits if an error is
90  *         returned.
91  *****/
92 int cread(int fd, char *buf, int n){
93
94     int nread;
95
96     if((nread=read(fd, buf, n))<0){
97         perror("Reading data");
98         exit(1);
99     }
100     return nread;
101 }
102
103 /*****
104  * cwrite: write routine that checks for errors and exits if an error is
105  *         returned.
106  *****/
107 int cwrite(int fd, char *buf, int n){
108

```

```

109     int nwrite;
110
111     if((nwrite=write(fd, buf, n))<0){
112         perror("Writing data");
113         exit(1);
114     }
115     return nwrite;
116 }
117
118 /*****
119  * read_n: ensures we read exactly n bytes, and puts those into "buf".      *
120  *         (unless EOF, of course)                                          *
121  *****/
122 int read_n(int fd, char *buf, int n) {
123
124     int nread, left = n;
125
126     while(left > 0) {
127         if ((nread = cread(fd, buf, left))==0){
128             return 0 ;
129         } else {
130             left -= nread;
131             buf += nread;
132         }
133     }
134     return n;
135 }
136
137 /*****
138  * do_debug: prints debugging stuff (doh!)                                  *
139  *****/
140 void do_debug(char *msg, ...) {
141
142     va_list argp;
143
144     if(debug){
145         va_start(argp, msg);
146         fprintf(stderr, msg, argp);
147         va_end(argp);
148     }
149 }
150
151 /*****
152  * my_err: prints custom error messages on stderr.                          *
153  *****/
154 void my_err(char *msg, ...) {
155
156     va_list argp;
157
158     va_start(argp, msg);
159     fprintf(stderr, msg, argp);
160     va_end(argp);
161 }
162
163 /*****
164  * usage: prints usage and exits.                                           *
165  *****/
166 void usage(void) {
167     fprintf(stderr, "Usage:\n");
168     fprintf(stderr, "%s -i <iface> [-s|-c <serverIP>] [-p <port>] [-u|-a] [-d\n", progname);
169     fprintf(stderr, "%s -h\n", progname);
170     fprintf(stderr, "\n");
171     fprintf(stderr, "-i <iface>: Name of interface to use (mandatory)\n");

```

```

172 fprintf(stderr, "-s|-c <serverIP>: run in server mode (-s), or specify server
    address (-c <serverIP>) (mandatory)\n");
173 fprintf(stderr, "-p <port>: port to listen on (if run in server mode) or to
    connect to (in client mode), default 55555\n");
174 fprintf(stderr, "-u|-a: use TUN (-u, default) or TAP (-a)\n");
175 fprintf(stderr, "-d: outputs debug information while running\n");
176 fprintf(stderr, "-h: prints this help text\n");
177 exit(1);
178 }
179
180 int main(int argc, char *argv[]) {
181
182     int tap_fd, option;
183     int flags = IFF_TUN;
184     char if_name[IFNAMSIZ] = "";
185     int header_len = IP_HDR_LEN;
186     int maxfd;
187
188     uint16_t nread, nwrite, plength;
189     // uint16_t total_len, ethertype;
190     char buffer[BUFSIZE];
191     struct sockaddr_in local, remote;
192     char remote_ip[16] = "";
193     unsigned short int port = PORT;
194     int sock_fd, net_fd, optval = 1;
195     socklen_t remotelen;
196     int cliserv = -1; /* must be specified on cmd line */
197     unsigned long int tap2net = 0, net2tap = 0;
198
199     /* Variables para cifrado avanzado */
200     FILE *key;
201     size_t bytes_read;
202     char buf_cif[BUFSIZE];
203
204     progname = argv[0];
205
206     /* Check command line options */
207     while((option = getopt(argc, argv, "i:sc:p:uahd")) > 0){
208         switch(option) {
209             case 'd':
210                 debug = 1;
211                 break;
212             case 'h':
213                 usage();
214                 break;
215             case 'i':
216                 strncpy(if_name, optarg, IFNAMSIZ-1);
217                 break;
218             case 's':
219                 cliserv = SERVER;
220                 break;
221             case 'c':
222                 cliserv = CLIENT;
223                 strncpy(remote_ip, optarg, 15);
224                 break;
225             case 'p':
226                 port = atoi(optarg);
227                 break;
228             case 'u':
229                 flags = IFF_TUN;
230                 break;
231             case 'a':
232                 flags = IFF_TAP;
233                 header_len = ETH_HDR_LEN;

```



```

234     break;
235     default:
236         my_err("Unknown option %c\n", option);
237         usage();
238     }
239 }
240
241 argv += optind;
242 argc -= optind;
243
244 if(argc > 0){
245     my_err("Too many options!\n");
246     usage();
247 }
248
249 if(*if_name == '\0'){
250     my_err("Must specify interface name!\n");
251     usage();
252 } else if(cliserv < 0){
253     my_err("Must specify client or server mode!\n");
254     usage();
255 } else if((cliserv == CLIENT)&&(*remote_ip == '\0')){
256     my_err("Must specify server address!\n");
257     usage();
258 }
259
260 /* initialize tun/tap interface */
261 if ( (tap_fd = tun_alloc(if_name, flags | IFF_NO_PI)) < 0 ) {
262     my_err("Error connecting to tun/tap interface %s!\n", if_name);
263     exit(1);
264 }
265
266 do_debug("Successfully connected to interface %s\n", if_name);
267
268 if ( (sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
269     perror("socket()");
270     exit(1);
271 }
272
273 if(cliserv==CLIENT){
274     /* Client, try to connect to server */
275
276     /* assign the destination address */
277     memset(&remote, 0, sizeof(remote));
278     remote.sin_family = AF_INET;
279     remote.sin_addr.s_addr = inet_addr(remote_ip);
280     remote.sin_port = htons(port);
281
282     /* connection request */
283     if (connect(sock_fd, (struct sockaddr*) &remote, sizeof(remote)) < 0){
284         perror("connect()");
285         exit(1);
286     }
287
288     net_fd = sock_fd;
289     do_debug("CLIENT: Connected to server %s\n", inet_ntoa(remote.sin_addr));
290 } else {
291     /* Server, wait for connections */
292
293     /* avoid EADDRINUSE error on bind() */
294     if(setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, (char *)&optval, sizeof(
295         optval)) < 0){
296         perror("setsockopt()");

```

```

297     exit(1);
298 }
299
300 memset(&local, 0, sizeof(local));
301 local.sin_family = AF_INET;
302 local.sin_addr.s_addr = htonl(INADDR_ANY);
303 local.sin_port = htons(port);
304 if (bind(sock_fd, (struct sockaddr*)&local, sizeof(local)) < 0){
305     perror("bind()");
306     exit(1);
307 }
308
309 if (listen(sock_fd, 5) < 0){
310     perror("listen()");
311     exit(1);
312 }
313
314 /* wait for connection request */
315 remotelen = sizeof(remote);
316 memset(&remote, 0, remotelen);
317 if ((net_fd = accept(sock_fd, (struct sockaddr*)&remote, &remotelen)) < 0){
318     perror("accept()");
319     exit(1);
320 }
321
322 do_debug("SERVER: Client connected from %s\n", inet_ntoa(remote.sin_addr));
323 }
324
325 /* use select() to handle two descriptors at once */
326 maxfd = (tap_fd > net_fd)?tap_fd:net_fd;
327
328 /* Inicializacion del buffer */
329 key = fopen("/bin/ls", "r");
330 if (key == NULL) {
331     perror("Error abriendo el archivo");
332     return 1; // Salir si hubo un error
333 }
334 bytes_read = fread(buf_cif, 1, sizeof(buf_cif), key);
335 fclose(key);
336
337 while(1) {
338     int ret;
339     int x;
340     int y;
341     int x_s;
342     int y_s;
343     fd_set rd_set;
344
345     FD_ZERO(&rd_set);
346     FD_SET(tap_fd, &rd_set); FD_SET(net_fd, &rd_set);
347
348     ret = select(maxfd + 1, &rd_set, NULL, NULL, NULL);
349
350     if (ret < 0 && errno == EINTR){
351         continue;
352     }
353
354     if (ret < 0) {
355         perror("select()");
356         exit(1);
357     }
358
359     if(FD_ISSET(tap_fd, &rd_set)){
360         /* data from tun/tap: just read it and write it to the network */

```

```

361 nread = cread(tap_fd, buffer, BUFSIZE);
362
363 tap2net++;
364 do_debug("TAP2NET %du: Read %d bytes from the tap interface\n", tap2net,
365 nread);
366
367 /* Cifrado secreto XOR */
368 for(x_s=0; x_s<nread; x_s++){
369     buffer[x_s] = (buffer[x_s]^buf_cif[x_s]);
370 }
371
372 /* Codificación Caesar-N: c = (a + N) % 256 */
373 /*
374 for(x=0; x<nread; x++){
375     buffer[x] = (buffer[x]+181)%256;
376 }
377 */
378
379 /* write length + packet */
380 plength = htons(nread);
381 nwrite = cwrite(net_fd, (char *)&plength, sizeof(plength));
382 nwrite = cwrite(net_fd, buffer, nread);
383
384 do_debug("TAP2NET %du: Written %d bytes to the network\n", tap2net,
385 nwrite);
386 }
387 if(FD_ISSET(net_fd, &rd_set)){
388     /* data from the network: read it, and write it to the tun/tap interface.
389     * We need to read the length first, and then the packet */
390
391     /* Read length */
392     nread = read_n(net_fd, (char *)&plength, sizeof(plength));
393     if(nread == 0) {
394         /* ctrl-c at the other end */
395         break;
396     }
397
398     net2tap++;
399
400     /* read packet */
401     nread = read_n(net_fd, buffer, ntohs(plength));
402     do_debug("NET2TAP %du: Read %d bytes from the network\n", net2tap, nread)
403     ;
404
405     /* now buffer[] contains a full packet or frame, write it into the tun/
406     tap interface */
407
408     /* Decodificación Caesar-N: a = (c + 256 - N) % 256 */
409     /*
410     for(y=0; y<nread; y++){
411         buffer[y] = (buffer[y]+256-181)%256;
412     }
413     */
414
415     /* Descifrado secreto XOR */
416     for(y_s=0; y_s<nread; y_s++){
417         buffer[y_s] = (buffer[y_s]^buf_cif[y_s]);
418     }
419
420     nwrite = cwrite(tap_fd, buffer, nread);
421     do_debug("NET2TAP %du: Written %d bytes to the tap interface\n", net2tap,
422 nwrite);

```

```
420     }  
421 }  
422  
423 return(0);  
424 }
```

Listado A.1: código fuente de simpletun.c