UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

VPN con simpletun

Trabajo Redes Corporativas

Grado en Ingeniería Informática

Autores:

Guillem Pruñonosa Soler

Daniel Soler Casesnoves

Grupo: 52

4º GII

Resumen

En la era de la información, cada vez la seguridad en la red es un aspecto más importante y crítico de la redes debido a las crecientes amenazas en la red. Una tecnología muy explotada para mejorar la seguridad son las VPN ya que permiten extender una LAN sobre Internet. Las VPN contienen mecanismos de ciberseguridad que proporcionan confidencialidad, autenticidad e integridad.

El principal objetivo de nuestro trabajo es implementar una VPN sencilla que proporcione confidencialidad. Nuestro objetivo es implementar dos versiones diferentes con dos sistemas de cifrado muy diferentes: César y AES. La VPN se implementará sobre un túnel virtual TUN.

Palabras clave: VPN, TUN, tunel, ciberseguridad, AES, cifrado, LAN, César, Internet.

Abstract

In the age of information, network security is becoming a more important and critical aspect of networks due to the increasing threats in the network. A widely exploited technology to improve security are VPNs since they allow a LAN to be extended over the Internet. VPNs contain cybersecurity mechanisms that provide confidentiality, authenticity, and integrity.

The main objective of our work is to implement a simple VPN that provides confidentiality. Our goal is to implement two different versions with two very different encryption systems: Cesar and AES. The VPN will be implemented over a virtual tunnel TUN.

Keywords : VPN, TUN, tunneling, cybersecurity, AES, encryption, LAN, Caesar, Network.

Tabla de contenidos

1. Introducción

En este trabajo vamos a poner en práctica y a exponer algunos de los conocimientos adquiridos en la asignatura de Redes Corporativas (RCO). En este trabajo en concreto nos centraremos en el diseño e implementación de una sencilla Red Privada Virtual (VPN), tanto en su infraestructura como en su seguridad.

Hoy en día, en la era de la información, la ciberseguridad es uno de los temas que más preocupan a los Estados y las empresas ya que cada vez, la información de la que disponen se encuentra más amenazada. Por ello, en este trabajo diseñaremos e implementaremos una herramienta red que permite realizar conexiones seguras entre dos puntos o entre dos redes privadas.

En la primera parte, nos dedicaremos a montar y configurar la red de la Figura 1 sobre la que funcionará nuestra VPN. Esta red será un red virtual que se hallará íntegramente en un PC. En dos hosts desplegaremos un dispositivo virtual que hará de Driver virtual, llamado TUN, con el que crearemos nuestro túnel que servirá de base para nuestra VPN.

En la segunda parte haremos pruebas con el túnel TUN creado, explicando sus fundamentos y los resultados de dichas pruebas.

En la última parte, implementaremos un cifrado de seguridad en nuestro túnel, con lo que ya podrá ser considerado una VPN. En esta parte implementaremos primero un cifrado sencillo de implementar y muy vulnerable, el cifrado Caesar, y luego haremos una segunda versión con un cifrado mucho más potente como es AES128.

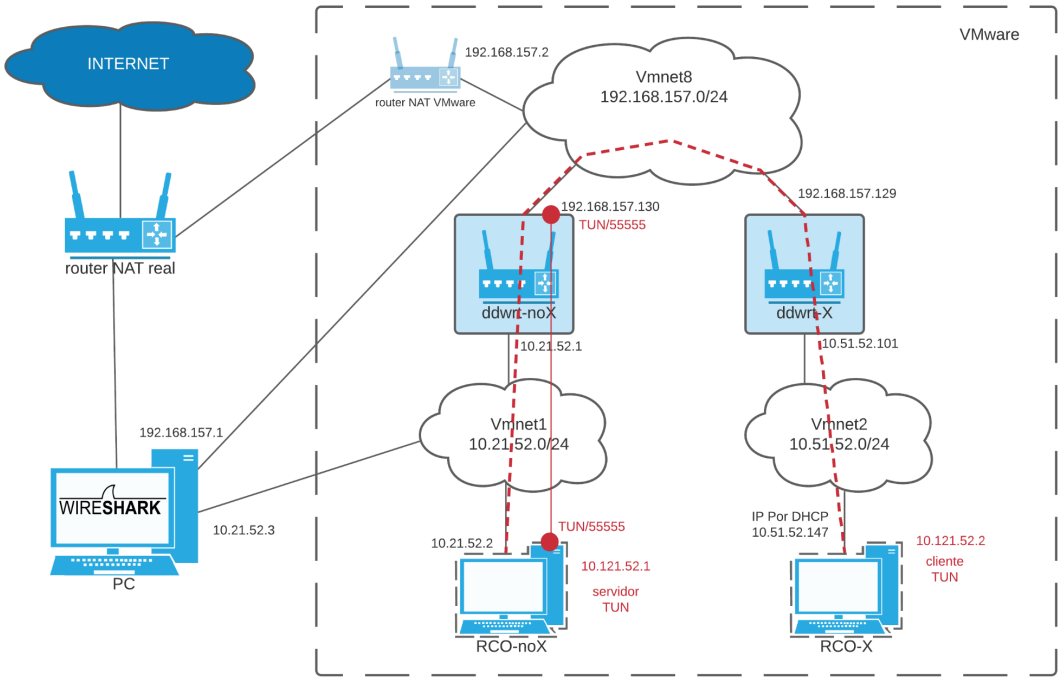

Figura 1: diseño de la red y la VPN implementada

2. Configuración

En este capitulo vamos a explicar la configuración de nuestra red, en la que configuraremos un host como servidor TUN y otro como cliente TUN, a parte de otras configuraciones necesarias para el correcto funcionamiento de nuestra VPN. Para realizar las siguientes configuraciones necesitamos tener las 4 máquinas encendidas.

Configuración de la redirección de puertos

A continuación procederemos a configurar la redirección de puertos para que el servidor TUN, que será RCO-noX, sea accesible desde fuera de la LAN ya que está detrás de un router NAT.

Añadiremos un port forwarding desde la interfaz web de DDWRT-noX, donde indicaremos que el tráfico que entre por el puerto TCP 55555 se dirija al puerto 55555 del Servidor simpletun RCO-noX.

Para llevar a cabo dichas modificaciones accederemos a NAT / QoS -> Port Forwarding y procederemos a añadir las modificaciones nombradas anteriormente como podemos ver en la Figura 2.

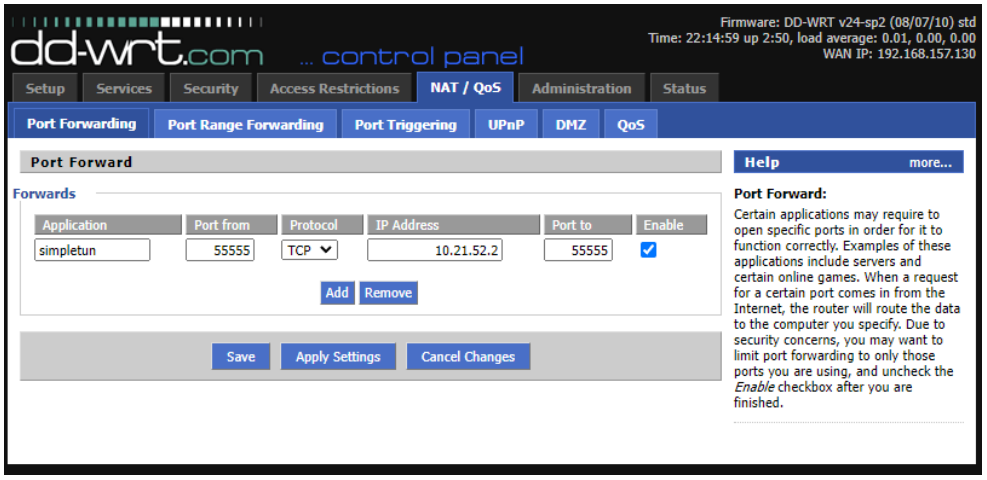

Figura 2: Creación de la redirección de puertos

Posteriormente procedemos a verificar que el redireccionamiento que hemos hecho en el paso anterior se haya creado de manera correcta con el comando iptables desde el terminal de ddwrt-noX.

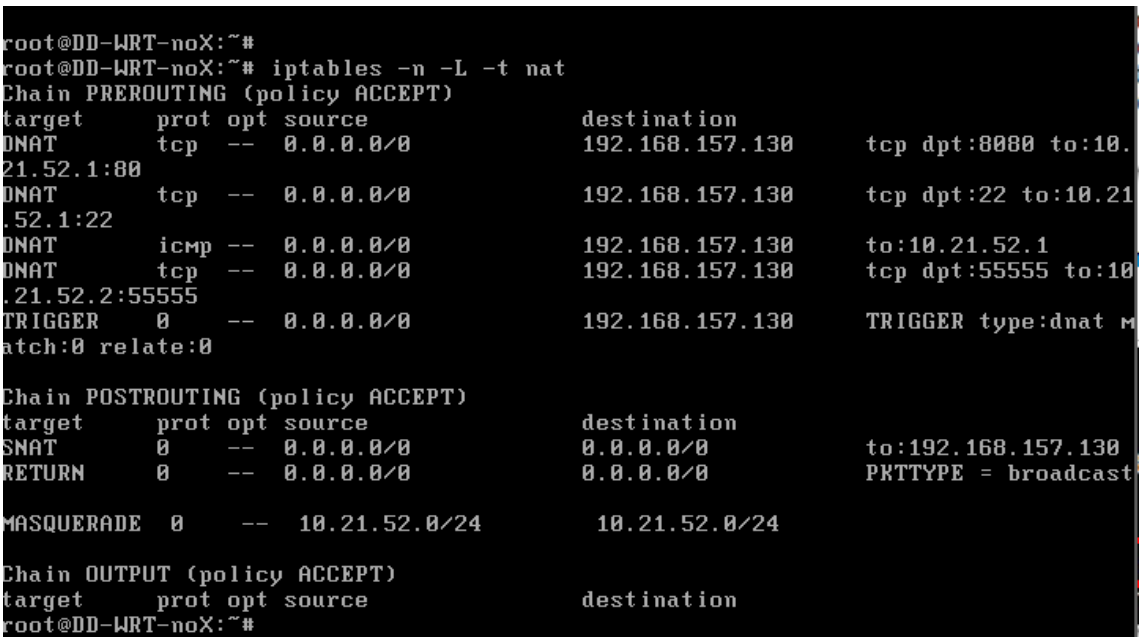

Figura 3: Orden iptables para ver la tabla nat

Tal y como esperábamos, en la Figura 3 podemos observar que todo lo que entre por TCP 55555 será enviado a 10.21.52.2:5555, es decir a nuestro servidor simpletun (RCO-noX).

Preparación CentOS para trabajar con tun/tap

A continuación vamos a descargarenos el archivo *simpletun.c* y lo compilaremos, abriendo la terminal e introduciendo los siguientes comandos:

cd # mkdir simpletun

cd simpletun/

wget https://redescorporativas.es/simpletun.c

make simpletun

Figura 4: Instalación simpletun.c

Tener en cuenta que para poder compilar *simpletun*, hay que tener instalado el compilador *gcc* tanto en el Servidor RCO-noX como en el cliente RCO-X simpletun.

Configuración túnel TUN/TAP

Vamos a proceder a montar un túnel *simpletun* entre el Servidor (RCO-noX) y el Cliente (RCO-X). Cabe resaltar que los pasos que vamos a describir a continuación son de caracter temporal por lo que cada vez que se reinicie el equipo tendremos que configurarlo de nuevo.

En primer lugar desde la máquina RCO-noX crearemos tun0 y lo configuraremos para que se conecte con la IP 10.121.52.1 y posteriormente lanzamos el servidor a la espera de que se conecte un cliente.

Figura 5 - Interficie tun 0 (servidor tun) en RCO-nox

En segundo lugar crearemos la interficie tun3 desde el host del cliente (RCO-X), con el objetivo de asociarlo a la IP local 10.121.52.2, consecuentemente el cliente se conecta con el servidor con la IP 192.168.157.130 (DDWRT-noX), recordar que el router tiene un *port forwarding* configurado en el puerto 55555. Tras conectarse entran ambos en bucle de escucha, el servidor RCO-noX en la interfaz tun0,y el cliente RCO-X en la interfaz tun3.

```
tun3: flags=4241<UP,POINTOPOINT,NOARP,MULTICAST>  mtu 1500
        inet 10.121.52.2  netmask 255.255.255.252  destination 10.121.52.2
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  txqueuelen 500  (UNS
PEC)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

[root@rco-x simpletun]#
```

Figura 6 - Interficie tun 3 (cliente tun) en rconox

Para probar el correcto funcionamiento del túnel hacemos un ping desde el cliente (RCO-X) al servidor 10.121.52.1.

```
[root@rco-x simpletun]# ping 10.121.52.1
PING 10.121.52.1 (10.121.52.1) 56(84) bytes of data.
64 bytes from 10.121.52.1: icmp_seq=1 ttl=64 time=44.3 ms
64 bytes from 10.121.52.1: icmp_seq=2 ttl=64 time=43.5 ms
64 bytes from 10.121.52.1: icmp_seq=3 ttl=64 time=45.3 ms
64 bytes from 10.121.52.1: icmp_seq=4 ttl=64 time=46.1 ms
64 bytes from 10.121.52.1: icmp_seq=5 ttl=64 time=45.4 ms
64 bytes from 10.121.52.1: icmp_seq=6 ttl=64 time=45.4 ms
64 bytes from 10.121.52.1: icmp_seq=7 ttl=64 time=42.3 ms
^C
--- 10.121.52.1 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6019ms
rtt min/avg/max/mdev = 42.338/44.609/46.055/1.215 ms
[root@rco-x simpletun]#
```

Figura 7 - ping al servidor para comprobar funcionamiento del túnel

Como vemos en la Figura 7, el resultado del *ping* ha sido satisfactorio. El camino que seguirán los paquetes será exclusivamente por el Túnel TUN, como podemos ver en la Figura 8. En el siguiente capitulo explicaremos con más detalle su funcionamiento.

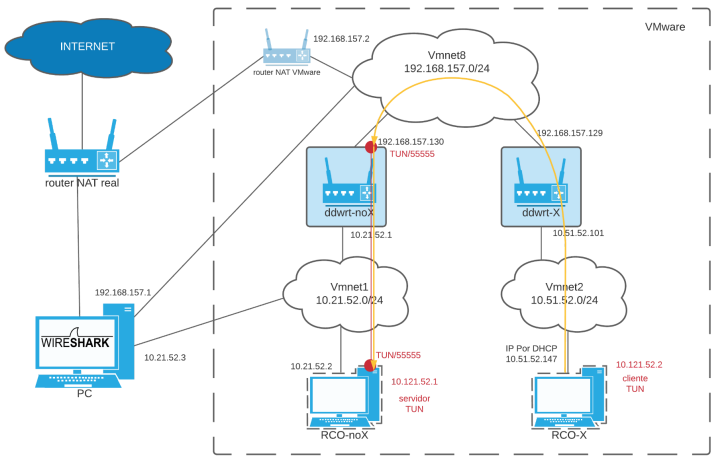

Figura 8: camino de los paquetes en el *ping* del cliente al servidor TUN

Túnel site-to-site

Llegados a este punto, después de comprobar mediante un ping el funcionamiento del túnel, tenemos configurado un túnel TUN entre dos puntos de 2 LANs diferentes RCO-noX y RCO-X, añadiremos las reglas de routing necesarias para que todas las máquinas de ambas LANs se comuniquen con cualquier otro host de la otra LAN.

Lo primero, será modificar las tablas de enrutamiento del servidor y el cliente TUN, es decir RCO-noX y RCO-X respectivamente.

```
[root@rco-nox simpletun]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.21.52.1      0.0.0.0         UG    100    0        0 ens37
10.21.52.0      0.0.0.0         255.255.255.0   U     100    0        0 ens37
10.121.52.0     0.0.0.0         255.255.255.252 U     0      0        0 tun0
[root@rco-nox simpletun]# ip route add 10.51.52.0/24 dev tun0
[root@rco-nox simpletun]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.21.52.1      0.0.0.0         UG    100    0        0 ens37
10.21.52.0      0.0.0.0         255.255.255.0   U     100    0        0 ens37
10.51.52.0      0.0.0.0         255.255.255.0   U     0      0        0 tun0
10.121.52.0     0.0.0.0         255.255.255.252 U     0      0        0 tun0
[root@rco-nox simpletun]#
```

Figura 9: camino de los paquetes en el *ping* del cliente al servidor TUN

```
[root@rco-x simpletun]# ip route add 10.21.52.0/24 dev tun3
[root@rco-x simpletun]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.51.52.101    0.0.0.0         UG    100    0        0 ens37
10.21.52.0      0.0.0.0         255.255.255.0   U     0      0        0 tun3
10.51.52.0      0.0.0.0         255.255.255.0   U     100    0        0 ens37
10.121.52.0     0.0.0.0         255.255.255.252 U     0      0        0 tun3
[root@rco-x simpletun]#
```

Figura 10 - Reglas añadidas en RCO-X

Sin embargo, para que funcione vamos a modificar estas reglas de enrutamiento que acabamos de añadir para que solo envíen el tráfico dirigido a la otra red por el túnel en caso de que el origen sea el mismo host. Esto lo hacemos para que el emisor del mensaje sea la interfaz LAN del host y no la interfaz del túnel.

Con estas nuevas reglas todo el tráfico que llega a cualquiera de los dos que tenga como destino la otra red, irá encaminado por la interfaz TUN correspondiente.

Lo siguiente es permitir que no solo estos dos hosts puedan comunicarse con la otra LAN por el túnel, sino que todos los hosts de ambas redes puedan.

Para ello, vamos a hacer que RCO-X y RCO-noX funcionen como routers interconectados por el túnel TUN, con la idea de que encaminen todo el tráfico entre ambas LANs.

Para ello debemos activar la opción de *forwarding* en ambos RCOs. Esta es una opción que proporciona el sistema Linux, y permite que cuando los RCOs 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. La orden en el terminal para activarlo es la siguiente:

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

También debemos crear tres nuevas reglas, la primera de ellas la añadiremos en el router ddwrt-noX, con el objetivo de que todo el tráfico que vaya dirigido a 10.51.52.0/24 salga por la puerta de enlace de la interfaz ens37 de nuestro servidor RCO-noX (10.21.52.2). Para ello añadimos la siguiente regla:

```
# ip route add 10.51.52.0/24 via 10.21.52.2
```

```
root@DD-WRT-noX:~# ip route add 10.51.52.0/24 via 10.21.52.2
root@DD-WRT-noX:~# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.157.2   0.0.0.0         255.255.255.255 UH    0      0        0 eth0
10.51.52.0      10.21.52.2      255.255.255.0   UG    0      0        0 br0
10.21.52.0      0.0.0.0         255.255.255.0   U     0      0        0 br0
192.168.157.0   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         192.168.157.2   0.0.0.0         UG    0      0        0 eth0
root@DD-WRT-noX:~#
```

Figura 11 - Reglas site-to-site ddwrt-noX

La segunda regla que hemos añadido se lleva a cabo en el router ddwrt-X. De manera similar a la explicada anteriormente, el tráfico dirigido hacia 10.21.52.0/24 su puerta de enlace correspondiente será la 10.51.52.147, la interfaz ens37. Podemos ver reflejada la nueva regla incorporada a la tabla de enrutamiento en la Figura 11

```
root@DD-WRT:~# ip route add 10.21.52.0/24 via 10.51.52.147
root@DD-WRT:~# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.157.2   0.0.0.0         255.255.255.255 UH    0      0        0 eth0
10.51.52.0      0.0.0.0         255.255.255.0   U     0      0        0 br0
10.21.52.0      10.51.52.147    255.255.255.0   UG    0      0        0 br0
192.168.157.0   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         192.168.157.2   0.0.0.0         UG    0      0        0 eth0
root@DD-WRT:~#
```

Figura 11 - Reglas site-to-site ddwrt-X

La última regla que vamos a añadir la añadimos a nuestro PC con el fin de que todo el tráfico que tenga como destin0 el 10.51.52.0/24, su puerta de enlace correspondiente será la 10.21.52.2.

3. Funcionamiento del Simpletun

Para alcanzar el objetivo de este trabajo es fundamental hacer uso del túnel *TUN/TAP*, conectando así dos redes privadas. En este caso en concreto haremos uso de las interfaces *TUN* con un cliente RCO-X y un servidor RCO-noX.

A continuación consideramos que es necesario explicar a grandes rasgos el funcionamiento de las interfaces de red *TUN* y *TAP*.

Tanto las interfaz *TUN* como *TAP* no gestionan ningún hardware, son interfaces a dispositivos software, es decir, *TUN/TAP* proporciona recepción y transmisión de datos para programas espaciales de usuario. Se puede considerar como un dispositivo punto a punto o ethernet.

TUN/TAP tiene dos modos de controlador. En primer lugar *TUN*, dispositivo punto a punto virtual de nivel tres, es decir trabajaremos con datagramas IP. En segundo lugar *TAP* , dispositivo ethernet virtual, trabajando así con datagramas ethernet.

Comprobación del túnel

Vamos a explicar el funcionamiento del túnel a partir de las pruebas anteriores.

La primera era un *ping* del cliente (10.121.52.2) al servidor TUN (10.121.52.1). Estos son los paquetes capturados:

1	0.000000000	fe80::250:56ff:…	ff02::fb	MDNS	180	Standard query 0x0000 PTR _ftp._tcp.local, "QM" question PTR _nfs._tcp.local, "QM" question PTR _a
2	0.000166605	10.51.52.147	224.0.0.251	MDNS	160	Standard query 0x0000 PTR _ftp._tcp.local, "QM" question PTR _nfs._tcp.local, "QM" question PTR _a
3	7.490004037	10.121.52.2	10.121.52.1	ICMP	1028	Echo (ping) request id=0x6a1a, seq=1/256, ttl=64 (reply in 4)
5	7.490373076	10.51.52.147	192.168.157.130	TCP	68	48028 → 55555 [PSH, ACK] Seq=1 Ack=1 Win=318 Len=2 TSval=4115645235 TSecr=1755839294
6	7.531902288	192.168.157.130	10.51.52.147	TCP	66	55555 → 48028 [ACK] Seq=1 Ack=3 Win=322 Len=0 TSval=1756046435 TSecr=4115645235
7	7.531935844	10.51.52.147	192.168.157.130	TCP	1094	48028 → 55555 [PSH, ACK] Seq=3 Ack=1 Win=318 Len=1028 TSval=4115645276 TSecr=1756046435
8	7.532608269	192.168.157.130	10.51.52.147	TCP	66	55555 → 48028 [ACK] Seq=1 Ack=1031 Win=338 Len=0 TSval=1756046436 TSecr=4115645276
9	7.532623509	192.168.157.130	10.51.52.147	TCP	68	55555 → 48028 [PSH, ACK] Seq=1 Ack=1031 Win=338 Len=2 TSval=1756046436 TSecr=4115645276
10	7.532630909	10.51.52.147	192.168.157.130	TCP	66	48028 → 55555 [ACK] Seq=1031 Ack=3 Win=318 Len=0 TSval=4115645277 TSecr=1756046436
11	7.533121231	192.168.157.130	10.51.52.147	TCP	1094	55555 → 48028 [PSH, ACK] Seq=3 Ack=1031 Win=338 Len=1028 TSval=1756046436 TSecr=4115645277
12	7.533132766	10.51.52.147	192.168.157.130	TCP	66	48028 → 55555 [ACK] Seq=1031 Ack=1031 Win=334 Len=0 TSval=4115645277 TSecr=1756046436
4	7.533187819	10.121.52.1	10.121.52.2	ICMP	1028	Echo (ping) reply id=0x6a1a, seq=1/256, ttl=64 (request in 3)

Figura 13 - Captura wireshark RCO-X

En primer lugar encontramos el paquete *ICMP request* con origen en el cliente y destino el servidor. El paquete estará compuesto solamente con IP e ICMP ya que no disponemos de ethernet porque es de nivel de 3 (IP) y lo mandamos desde la interfaz

tun0, dicho paquete llega a nuestra aplicación *simpletun* cliente a través del túnel, a cual lo va a encapsular en un paquete TCP con destino el puerto 55555 de DDWRT-noX y lo retransmitirá a través de la red en un paquete Ethernet hacia el router DDWRT-X.

Podemos corroborar que el paquete está correctamente encapsulado porque si sumamos el tamaño del paquete ICMP más el IP (1028+66) es igual al tamaño del paquete ya encapsulado enviado via Ethernet. (Figura 13)

Respecto a los paquetes nº 5 (Figura 13) antes del envío del paquete encapsulado, envía el tamaño del paquete IP+ICMP para que posteriormente pueda ser desencapsulado en el destino.

Por último, podemos observar que a cada envío de paquete TCP recibimos un paquete ACK, confirmando que ha llegado de manera correcta el paquete a su destino.

Siguiendo con las pruebas, a continuación vamos a verificar el correcto funcionamiento de las reglas de routing añadidas anteriormente en la Figura 10. Para ello hacemos un ping desde el cliente *RCO-X* hacia la *IP* privada de *RCO-noX*.

```
 1 0.000000000   10.121.52.2       10.21.52.2        ICMP      84 Echo (ping) request  id=0x6ba8, seq=1/256, ttl=64 (reply in 2)
 3 0.000879321   10.51.52.147      192.168.157.130   TCP       68 48028 → 55555 [PSH, ACK] Seq=1 Ack=1 Win=382 Len=2 TSval=41157875
 4 0.048143289   192.168.157.130   10.51.52.147      TCP       66 55555 → 48028 [ACK] Seq=1 Ack=3 Win=386 Len=0 TSval=1756188745 TS
 5 0.048180654   10.51.52.147      192.168.157.130   TCP      150 48028 → 55555 [PSH, ACK] Seq=3 Ack=1 Win=382 Len=84 TSval=4115787
 6 0.048876218   192.168.157.130   10.51.52.147      TCP       66 55555 → 48028 [ACK] Seq=1 Ack=87 Win=386 Len=0 TSval=1756188746 T
 7 0.048979650   192.168.157.130   10.51.52.147      TCP       68 55555 → 48028 [PSH, ACK] Seq=1 Ack=87 Win=386 Len=2 TSval=1756188
 8 0.049009590   10.51.52.147      192.168.157.130   TCP       66 48028 → 55555 [ACK] Seq=87 Ack=3 Win=382 Len=0 TSval=4115787588 T
 9 0.049630503   192.168.157.130   10.51.52.147      TCP      150 55555 → 48028 [PSH, ACK] Seq=3 Ack=87 Win=386 Len=84 TSval=175618
10 0.049643862   10.51.52.147      192.168.157.130   TCP       66 48028 → 55555 [ACK] Seq=87 Ack=87 Win=382 Len=0 TSval=4115787589
 2 0.049742968   10.21.52.2        10.121.52.2       ICMP      84 Echo (ping) reply    id=0x6ba8, seq=1/256, ttl=64 (request in 1)
```

Figura 14: tráfico capturado en el ping de RCO-X a 10.21.52.2

El funcionamiento es el mismo que en el anterior ping a excepción de que después de llegar al simpletun del servidor y ser desencapsulado, ve que tun0 no es la destinación final y mirando la tabla de enrutamiento, encamina el paquete ICMP a su destino final, en este caso la interfaz ens37 (10.21.52.2) de RCO-noX.

También hemos hecho otra prueba, modificando la regla de enrutamiento añadida en el paso anterior (Figura 10). La modificación es que los dos hosts RCO solo encaminen el tráfico destinado a la otra LAN si el paquete proviene de mismo RCO. En este caso resultado es el mismo que en la anterior prueba, salvo en la IP de origen que es la dirección de RCO-X en la LAN.

```
No.  Time          Source            Destination       Protoc Length Info
     1 0.000000000 VMware_33:ba:13   Broadcast         ARP      60 Who has 10.51.52.147? Tell 10.51.52.101
     2 0.000013350 VMware_38:60:9c   VMware_33:ba:13   ARP      42 10.51.52.147 is at 00:50:56:38:60:9c
     3 0.000708768 34.210.39.83      10.51.52.147      TLS…     85 Application Data
     4 0.015564541 10.51.52.147      34.210.39.83      TLS…     89 Application Data
     5 0.015982348 34.210.39.83      10.51.52.147      TCP      60 443 → 32806 [ACK] Seq=32 Ack=36 Win=64240 Len=0
     6 5.111378129 VMware_38:60:9c   VMware_33:ba:13   ARP      42 Who has 10.51.52.101? Tell 10.51.52.147
     7 5.111681738 VMware_33:ba:13   VMware_38:60:9c   ARP      60 10.51.52.101 is at 00:50:56:33:ba:13
     8 5.360233250 10.51.52.147      10.21.52.2        ICMP   1028 Echo (ping) request  id=0x6f5f, seq=1/256, ttl=64 (reply in 9)
    10 5.360816260 10.51.52.147      192.168.157.130   TCP      68 48028 → 55555 [PSH, ACK] Seq=1 Ack=1 Win=382 Len=2 TSval=4116367450 TSecr=1756241555
    11 5.369673350 192.168.157.130   10.51.52.147      TCP      66 55555 → 48028 [ACK] Seq=1 Ack=3 Win=386 Len=0 TSval=1756768613 TSecr=4116367450
    12 5.369714829 10.51.52.147      192.168.157.130   TCP    1094 48028 → 55555 [PSH, ACK] Seq=3 Ack=1 Win=382 Len=1028 TSval=4116367459 TSecr=1756768613
    13 5.370251180 192.168.157.130   10.51.52.147      TCP      66 55555 → 48028 [ACK] Seq=1 Ack=1031 Win=402 Len=0 TSval=1756768614 TSecr=4116367459
    14 5.370439609 192.168.157.130   10.51.52.147      TCP      68 55555 → 48028 [PSH, ACK] Seq=1 Ack=1031 Win=402 Len=2 TSval=1756768614 TSecr=4116367459
    15 5.370453902 10.51.52.147      192.168.157.130   TCP      66 48028 → 55555 [ACK] Seq=1031 Ack=3 Win=382 Len=0 TSval=4116367460 TSecr=1756768614
    16 5.371169957 192.168.157.130   10.51.52.147      TCP    1094 55555 → 48028 [PSH, ACK] Seq=3 Ack=1031 Win=402 Len=1028 TSval=1756768615 TSecr=4116367460
    17 5.371184228 10.51.52.147      192.168.157.130   TCP      66 48028 → 55555 [ACK] Seq=1031 Ack=1031 Win=398 Len=0 TSval=4116367461 TSecr=1756768615
     9 5.373912992 10.21.52.2        10.51.52.147      ICMP   1028 Echo (ping) reply    id=0x6f5f, seq=1/256, ttl=64 (request in 8)
```

Figura 15 - Captura wireshark posterior a la modificación de las tablas de enrutamiento

Como vemos en la captura de Wireshark, ahora la dirección de origen es 10.51.52.147, la IP de la interfaz ens37, que está conectada a la LAN, de RCO-X.

Túnel site-to-site

Como hemos dicho en el capitulo anterior lo que nos interesa para nuestra VPN es que todos los hosts de una LAN puedan comunicarse con cualquier host de la otra LAN a través del túnel. Los pasos que hemos seguido para configurar esta parte están descritos en el capitulo antes referenciado.

Una vez realizados todos los cambios vamos a realizar una prueba: *ping* de DDWRT-X a DDWRT-noX.

```
root@DD-WRT:~# ping 10.21.52.1
PING 10.21.52.1 (10.21.52.1): 56 data bytes
64 bytes from 10.21.52.1: seq=0 ttl=62 time=4.927 ms
64 bytes from 10.21.52.1: seq=1 ttl=62 time=57.040 ms
64 bytes from 10.21.52.1: seq=2 ttl=62 time=48.902 ms
64 bytes from 10.21.52.1: seq=3 ttl=62 time=49.030 ms

--- 10.21.52.1 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 4.927/39.974/57.040 ms
root@DD-WRT:~# eth0: link down
```

Figura 16: Resultado del ping entre routers DDWRT

5	-0.000024926	10.51.52.101	10.21.52.1	ICMP	98	Echo (ping) request id=0x2b0e, seq=0/0, ttl=64 (reply in 14)
1	0.000000000	10.51.52.101	10.21.52.1	ICMP	84	Echo (ping) request id=0x2b0e, seq=0/0, ttl=63 (reply in 2)
6	0.000790333	10.51.52.147	192.168.157.130	TCP	68	48138 → 55555 [PSH, ACK] Seq=1 Ack=1 Win=229 Len=2 TSval=2837209
7	0.058981870	192.168.157.130	10.51.52.147	TCP	66	55555 → 48138 [ACK] Seq=1 Ack=3 Win=227 Len=0 TSval=3855712323 T
8	0.059009132	10.51.52.147	192.168.157.130	TCP	150	48138 → 55555 [PSH, ACK] Seq=3 Ack=1 Win=229 Len=84 TSval=283726
9	0.059598903	192.168.157.130	10.51.52.147	TCP	66	55555 → 48138 [ACK] Seq=1 Ack=87 Win=227 Len=0 TSval=3855712324
10	0.059942991	192.168.157.130	10.51.52.147	TCP	68	55555 → 48138 [PSH, ACK] Seq=1 Ack=87 Win=227 Len=2 TSval=385571
11	0.059951876	10.51.52.147	192.168.157.130	TCP	66	48138 → 55555 [ACK] Seq=87 Ack=3 Win=229 Len=0 TSval=2837209336
12	0.060351209	192.168.157.130	10.51.52.147	TCP	150	55555 → 48138 [PSH, ACK] Seq=3 Ack=87 Win=227 Len=84 TSval=38557
13	0.060361648	10.51.52.147	192.168.157.130	TCP	66	48138 → 55555 [ACK] Seq=87 Ack=87 Win=229 Len=0 TSval=2837209336
2	0.060402156	10.21.52.1	10.51.52.101	ICMP	84	Echo (ping) reply id=0x2b0e, seq=0/0, ttl=63 (request in 1)
14	0.060412341	10.21.52.1	10.51.52.101	ICMP	98	Echo (ping) reply id=0x2b0e, seq=0/0, ttl=62 (request in 5)

Figura 17: Resultado del ping entre routers DDWRT

Estos han sido los resultados de la prueba, como vemos funciona a la perfección. Como se ve a simple vista, la gran diferencia es que ahora capturamos 2 paquetes ICMP Request y 2 ICMP Reply. Vamos a mostrar una imagen donde se ve el recorrido de las tramas:

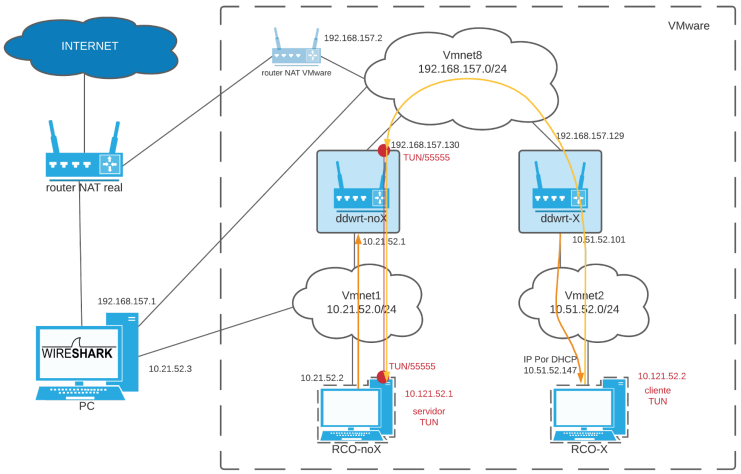

Figura 18: Resultado del ping entre routers DDWRT

En esta figura se aprecia perfectamente el "camino" que sigue el ICMP Request desde DDWRt-X a DDWRT-noX. Hemos indicado en color naranja los paquetes que no van por el túnel TUN. Estos son el paquete 5 y 14 en nuestra captura Wireshark (Figura 17). El camino amarillo es el que siguen los paquetes en el túnel, en este caso los TCP, excepto los de tamaño 66 que son los ACK de respuesta y tienen el campo de datos TCP vacío.

El primero de los paquetes ICMP request es el que RCO-X captura en su interfaz eth0, el ICMP request origen, que el DDWRT-X envia a DDWRT-noX. Como hemos visto antes, DDWRT-X tiene configurado en la tabla de enrutamiento que para todo el tráfico dirigido a la 10.21.52.0/24 (DDWRT-noX está en esta red) la puerta de enlace sea la interfaz ens37 (10.51.52.147) de RCO-X. Este paquete contiene cabecera Ethernet ya que se envía a Nivel 2, desde la interfaz Ethernet de DDWRT-X a su par en RCO-X.

Tal y como lo tenemos configurado, como hemos descrito anteriormente, los hosts RCO-X y RCO-noX se comportaran como routers de nuestra VPN, ya que encaminan todo el tráfico que entra y sale del túnel TUN.

En este caso, al recibir RCO-X el ICMP request, decide enviar el paquete por su interfaz *tun3*, hacia el túnel, ya que es lo que le marca su tabla de enrutamiento, que todo el tráfico con destino 10.21.52.0/24 vaya por el túnel TUN.

Los paquetes enviados por el túnel realmente son enviados encapsulados en segmentos TCP y enviados por la interfaz física *ens37* hacia fuera de la red, hacia el router DDWRT-X, el cual contesta con ACKs, por eso los paquetes de 66B. Luego DDWRT-X ya lo envía por entrega directa al puerto 55555 DDWRT-noX, que al tener activo el *port-forwarding* enviará los paquetes recibidos en el puerto 55555 al mismo puerto de RCO-noX, el cuál lo desempaquetará y enviará el datagrama IP+ICMP Request a la

interfaz *tun0* y luego conforme a la tabla de enrutamiento, será enviado a su destino final, DDWRT-noX, a través de la interfaz física *ens37*

Es importante destacar que antes de recibir y procesar el paquete el ICMP, el receptor *simpletun* debe conocer el tamaño del paquete que recibe. Es por ello que el emisor envía primero un paquete TCP indicando el tamaño del paquete, este es el paquete de 68B de la captura.

4. Cifrado VPN

En esta parte vamos a implementar dos sistemas de cifrado para securizar nuestro túnel TUN y convertirlo así en una Red Privada Virtual (VPN) segura. Primero implementaremos un sencillo sistema de cifrado llamada Caesar, y luego implementaremos un sistema más avanzado y mucho más seguro llamado AES.

Cifrado Caesar

El cifrado César (Caesar en Latín) o cifrado por desplazamiento es un tipo de cifrado por sustitución, es decir, unidades de texto plano son sustituidas con texto cifrado siguiendo un sistema regular. (Wikipedia, 2021)

El cifrado César consiste en un cifrado en el que un carácter ASCII en el texto original es reemplazada por otro carácter ASCII que se encuentra un número fijo de posiciones más adelante. (Mishra, 2021)

El receptor descifra el texto realizando la sustitución inversa.

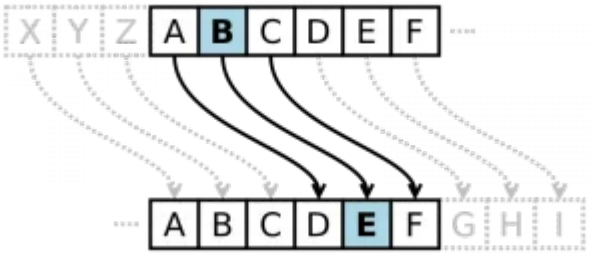

Figura 19: esquema del funcionamiento del cifrado César

Para aplicar este cifrado a nuestro *simpletun* vamos a crear dos funciones, una que sea de cifrar y otra de descifrar. La primera se ejecutará antes de la interfaz virtual TUN envie el paquete a la interfaz de red Ethernet, es decir antes de que lo envie por el túnel TUN. La segunda seguirá los pasos inversos, y descifrará el paquete cuando llegue un paquete por el túnel a la interfaz TUN.

Los metodos implementados en *simpletun* son los siguientes:

```c
void caesar_enc(char *buf,int n){
   char ch;
   int i;

   for(i = 0; i<n; ++i){
      ch = buf[i];
      buf[i] = (ch + 52) % 256;
   }
}
```

Figura 20: funcion del cifrado César

```c
void caesar_dec(char *buf, int n){
   char ch;
   int i;

   for(i = 0; i<n; ++i){
      ch = buf[i];
      buf[i] = (ch + 256 - 52) % 256;
   }
}
```

Figura 21: funcion del descifrado de código César

A ambas funciones se les pasa un puntero al *buffer* donde se almacena el texto a cifrar, en nuestro caso un paquete IP, con su cabecera y con los datos que lleva encapsulado; y una variable *n* que es el tamaño del paquete que se va a cifrar o descifrar. El resultado de las dos funciones se sobreescribe en el *buffer* de entrada.

Estas funciones se invocan desde el mismo programa *simpletun* cuando tanto el cliente como el servidor esperan, con un *while* (Figura 22), a que les llegue un paquete, ya sea desde la red (input) o desde la interfaz TUN (output).

```c
  if(FD_ISSET(tap_fd, &rd_set)){
    /* data from tun/tap: just read it and write it to the network */

    nread = cread(tap_fd, buffer, BUFSIZE); // leemos de la interfaz tun
    caesar_enc(buffer, nread); // ciframos el contenido del buffer

    tap2net++;
    do_debug("TAP2NET %lu: Read %d bytes from the tap interface\n", tap2net, nread);

    /* write length + packet */
    plength = htons(nread); // little endian -> big-endian
      // enviamos el tamaño del mensaje a enviar en formato big-endian (usado en TCP/IP)
    nwrite = cwrite(net_fd, (char *)&plength, sizeof(plength));
    nwrite = cwrite(net_fd, buffer, nread); // enviamos el mensaje cifrado a la red

    do_debug("TAP2NET %lu: Written %d bytes to the network\n", tap2net, nwrite);
  }

  if(FD_ISSET(net_fd, &rd_set)){
    /* data from the network: read it, and write it to the tun/tap interface.
     * We need to read the length first, and then the packet */

    /* Read length */
      // leemos el tamaño del paquete que nos ha llegado de la interfaz red
    nread = read_n(net_fd, (char *)&plength, sizeof(plength));
    if(nread == 0) { // si tamaño del contenido = 0 -> termina el programa
      /* ctrl-c at the other end */
      break;
    }

    net2tap++;

    /* read packet */
    nread = read_n(net_fd, buffer, ntohs(plength)); // leemos el paquete de la interfaz red
    do_debug("NET2TAP %lu: Read %d bytes from the network\n", net2tap, nread);

    caesar_dec(buffer, nread); // desciframos el paquete cifrado procedente de la red

    /* now buffer[] contains a full packet or frame, write it into the tun/tap interface */
    nwrite = cwrite(tap_fd, buffer, nread); // enviamos el mensaje plano por la interfaz TUN
    do_debug("NET2TAP %lu: Written %d bytes to the tap interface\n", net2tap, nwrite);
  }
}
```

Figura 22: fragmento del codigo donde se ejecuta el cifrado César

Como vemos en el código, se cifra el mensaje si viene de la interfaz TUN y se dirige a la interfaz red.

Invocamos a la función *caesar_enc* después de leer el paquete, calcular el tamaño de este y almacenarlo en *nread*. A continuación, tras cifrar el mensaje plano, enviamos primero un paquete con el tamaño que va a tener el paquete cifrado, y luego lo enviamos hacia la interfaz Ethernet con la función *cwrite*.

Y viceversa cuando el paquete viene de la red, solo que en este caso se descifra. La secuencia es la siguiente: leemos primero el tamaño del paquete (*plength*), luego leemos el paquete volcando el contenido en el *buffer* y guardandonos el tamaño del contenido en *nread*. Finalmente llamamos a *caesar_dec* para que descifre el mensaje cifrado de tamaño *nread*, volcándolo de nuevo en *buffer* y enviándolo por la interfaz TUN.

Vamos a realizar una prueba con tal de mostrar el funcionamiento de nuestro código de cifrado César. La prueba consistirá, como en una de las pruebas anteriores, en realizar un *ping* entre los routers DDWRT, en concreto desde DDWRT-X (10.51.52.101) a DDWRT-noX (10.21.52.1).

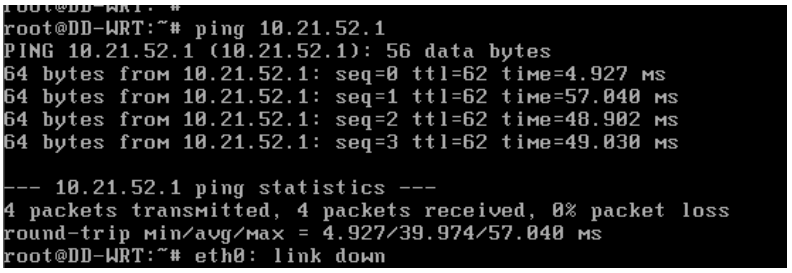

Figura 23: Resultado del ping entre routers DDWRT

Los resultados capturados en RCO-X (tanto en eth0 como en tun3) ordenados por tiempo son los siguientes:

8	15.446010992	10.51.52.101	10.21.52.1	ICMP	98	Echo (ping) request id=0x0317, seq=0/0, ttl=64 (reply in 17)
18	15.446054228	10.51.52.101	10.21.52.1	ICMP	84	Echo (ping) request id=0x0317, seq=0/0, ttl=63 (reply in 19)
9	15.446184966	10.51.52.147	192.168.157.130	TCP	68	34910 → 55555 [PSH, ACK] Seq=1 Ack=1 Win=29312 Len=2 TSval=3469
10	15.447415076	192.168.157.130	10.51.52.147	TCP	66	55555 → 34910 [ACK] Seq=1 Ack=3 Win=29056 Len=0 TSval=421768745
11	15.447434775	10.51.52.147	192.168.157.130	TCP	150	34910 → 55555 [PSH, ACK] Seq=3 Ack=1 Win=29312 Len=84 TSval=346
12	15.448426045	192.168.157.130	10.51.52.147	TCP	66	55555 → 34910 [ACK] Seq=1 Ack=87 Win=29056 Len=0 TSval=42176874
13	15.449177635	192.168.157.130	10.51.52.147	TCP	68	55555 → 34910 [PSH, ACK] Seq=1 Ack=87 Win=29056 Len=2 TSval=421
14	15.449192568	10.51.52.147	192.168.157.130	TCP	66	34910 → 55555 [ACK] Seq=87 Ack=3 Win=29312 Len=0 TSval=34693740
15	15.450193200	192.168.157.130	10.51.52.147	TCP	150	55555 → 34910 [PSH, ACK] Seq=3 Ack=87 Win=29056 Len=84 TSval=42
16	15.450201598	10.51.52.147	192.168.157.130	TCP	66	34910 → 55555 [ACK] Seq=87 Ack=87 Win=29312 Len=0 TSval=3469374
19	15.450319505	10.21.52.1	10.51.52.101	ICMP	84	Echo (ping) reply id=0x0317, seq=0/0, ttl=63 (request in 18)
17	15.450342728	10.21.52.1	10.51.52.101	ICMP	98	Echo (ping) reply id=0x0317, seq=0/0, ttl=62 (request in 8)

Figura 24: Tramas capturadas en el *ping* entre routers

Como vemos en las tramas capturadas tenemos 2 paquetes ICMP request, 8 TCP y 2 ICMP reply.

El primero de los paquetes ICMP request es el que RCO-X captura en su interfaz eth0, el ICMP request origen, que el DDWRT-X envia a DDWRT-noX. El segundo paquete, que vemos aquí abajo, es un paquete IP, que se captura en la interfaz TUN antes de ser enviado de nuevo a la red. El funcionamiento del túnel TUN con detalle ya lo hemos explicado anteriormente.

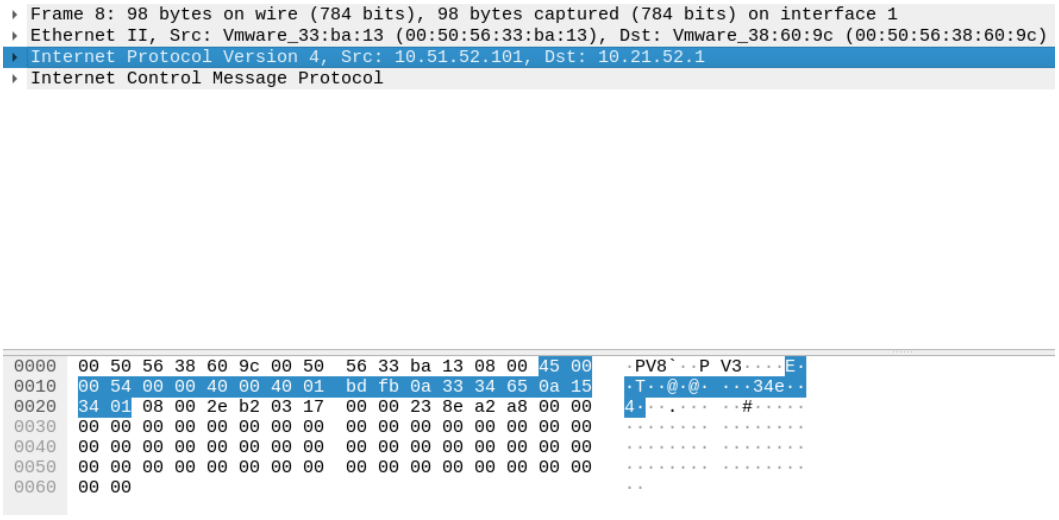

Figura 25: ICMP echo request sin cifrar de DDWRT-X a DDWRT-noX capturado por RCO-X en etho

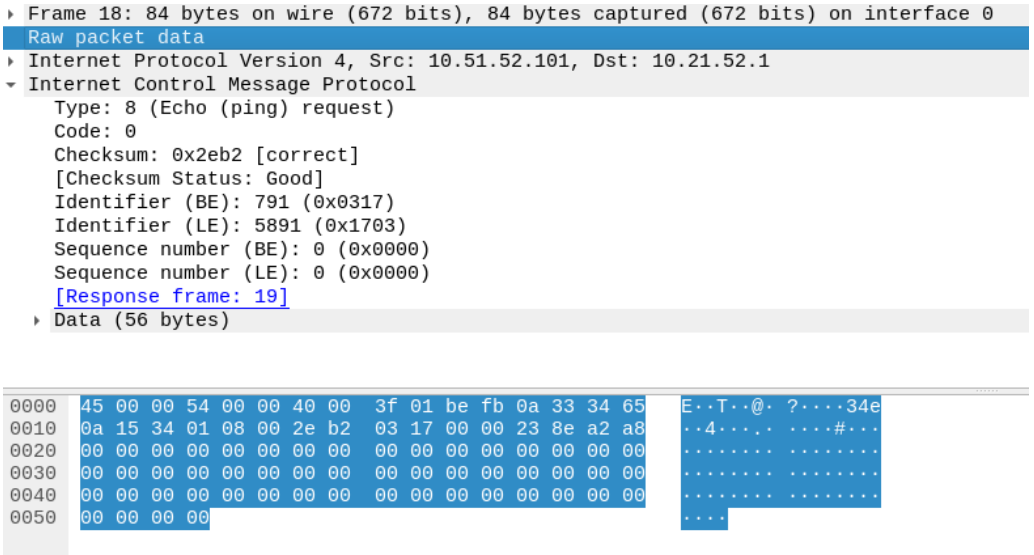

Figura 26: ICMP echo request sin cifrar de DDWRT-X a DDWRT-noX capturado por RCO-X en tun3

Estos 2 paquetes aún no están cifrados ya que DDWRT-X lo ha enviado a RCO-X en formato plano. Estos dos paquetes IP-ICMP son idénticos salvo por el decremento del TTL en el segundo (ver en Figura 26) .

Ahora, el paquete TCP capturado número 11 en la Figura 24, que tiene de tamaño 150,y que capturamos posteriormente a los ICMP Request, es el que enviamos tras cifrarlo.

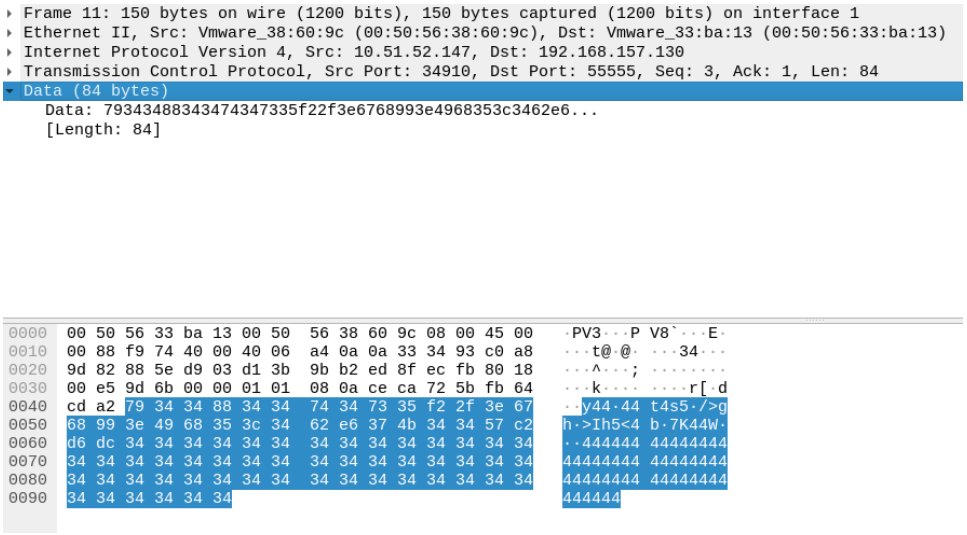

Figura 27: ICMP echo request cifrado de DDWRT-X a DDWRT-noX capturado por RCO-X en tun3

Como vemos en Figura X, el paquete está formado por Ethernet-IP-TCP- TCP Data. Este paquete se captura en la interfaz eth0 (interface 1), por ello es un paquete que está completo. Como podemos observar también, en el campo de datos de TCP hay encapsulado un paquete IP-ICMP (84B), este paquete es el de la Figura 26, pero esta vez cifrado por César.

Se puede apreciar el cifrado resaltado en azul. Como modo de ejemplo, podemos coger el primer y último número en Hexadecimal y ver su conversión de texto plano a cifrado. El primer número es el 0x45, y sumándole el desplazamiento fijo, 0x34 (52 en Hex), da 0x79. Y en cuanto al último número, aún más simple, 0x00 + 0x34 = 0x34. Como vemos estos son el primero y el último número de Data del paquete.

Como pequeño detalle, en el caso de César, coincide el tamaño del texto plano y el del cifrado. Puede parecer un detalle sin importancia pero como veremos en la siguiente técnica de cifrado, esto no siempre es así. En este caso, el tamaño del paquete de datos que envía por la red el *simpletun* en un paquete TCP antes de enviar los datos cifrados coincide con el tamaño del paquete IP-ICMP de la Figura 26. En este caso, 84 Bytes.

En la captura no se puede observar ya que no ocurre en la misma red, pero el paquete cifrado se descifra una vez llegue a la interfaz TUN de RCO-noX, para luego enviarlo en formato plano el ICMP request original a DDWRT-noX.

Como hemos explicado antes, todos los paquetes TCP son paquetes que van por nuestra VPN, y como podemos deducir en la Figura 18, el otro paquete TCP de 150B, es la respuesta del ICMP cifrada de la misma forma pero en el sentido contrario. La respuesta seguirá el camino contrario, pasando por el túnel TUN y descifrándose en el RCO-X antes de ser enviado al destino, DDWRT-X.

Cifrado AES128

Advanced Encryption Standard (AES), conocido también como Rijndael debido a sus creadores (Joan Daemen y Vincent Rijmen), es un esquema de cifrado de bloques. El cifrado de bloque es un tipo de cifrado de clave simétrica que opera en grupos de bits de longitud fija, llamados bloques, aplicándose una transformación invariante. (Wikipedia, 2021)

Hemos escogido AES como método de cifrado porque se trata de uno de los sistemas más robustos conocidos y más utilizados tanto por empresas como por entidades estatales. (López, 2021)

El funcionamiento de AES en pseudocódigo es el siguiente:

1. Expansión de la clave usando el esquema de claves de Rijndael [1]
2. Etapa inicial: *AddRoundKey:* cada byte del *state*[2] es combinado con un byte del *round-key* a través de un XOR bit a bit.
3. Varias rondas iterativas de:
    a. *SubBytes*: sustitución no lineal donde cada byte es reemplazado con otro de acuerdo a una tabla de búsqueda
    b. *ShiftRows*: transposición donde cada fila del «state» es rotada de manera cíclica un número determinado de veces.
    c. *MixColumns*: operación de mezclado que opera en las columnas del «state», combinando los cuatro bytes en cada columna usando una transformación lineal.
    d. *AddRoundKey*
4. Iterar varias veces *SubBytes*, *ShiftRows* y *AddRoundKey* varias veces.

En la siguiente figura ilustramos los pasos anteriores mencionados.

[1] https://en.wikipedia.org/wiki/AES_key_schedule
[2] AES opera en una matriz de 4×4 bytes, llamada state

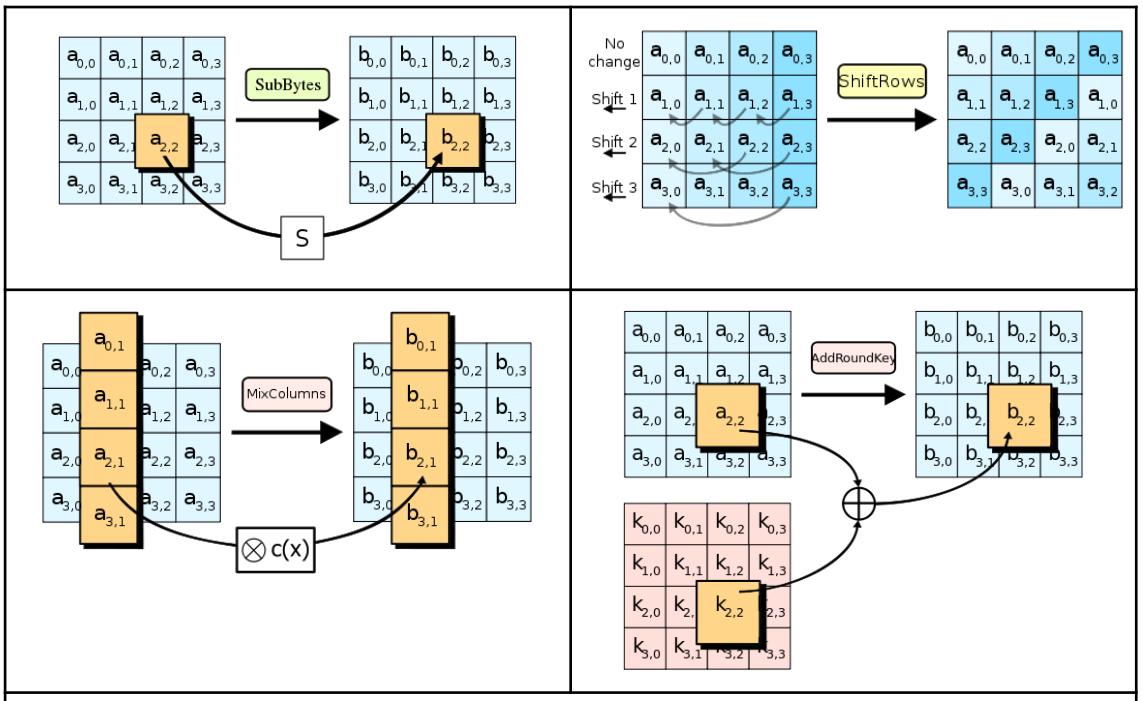

Figura28 : Proceso de cifrado de AES

Sin embargo, AES digamos que solo realiza la transformación criptográfica de un grupo de bits de longitud fija, un bloque. En nuestro caso vamos a cifrar por bloques de 128bits, pero existe AES192 y AES256, que cifran bloques de 192 y 256 bits respectivamente y son aún más robustos. (Wikipedia, 2021)

Por lo tanto para hacer transformaciones criptográficas de una cantidad de datos mayor a un bloque necesitamos lo que se denomina “modo de operación”. Un modo de operación describe cómo aplicar repetidamente una operación de cifrado en un bloque simple para la transformación segura de cantidades de datos mayores que un bloque. Los modos de cifrado por bloques operan con bloques completos, por lo que es necesario que la última parte del bloque sea rellenada, en caso de que su tamaño sea menor que el de los anteriores a él.

La mayoría de los modos requiere una secuencia binaria única, usualmente llamada vector de inicialización (IV), para cada operación de encriptación. El IV no tiene por qué repetirse y para algunos modos es aleatorio. El vector de inicialización se utiliza para asegurar que se generen textos cifrados distintos, aun cuando sea el mismo texto el que se encripte varias veces y con la misma clave.

Nosotros hemos usado CBC como modo de operación. En el modo CBC (cipher-block chaining), antes de ser cifrado, a cada bloque de texto se le aplica una operación XOR con el bloque previo ya cifrado. De este modo, cada bloque cifrado depende de todos los bloques de texto claros usados hasta ese punto. Además, para hacer cada mensaje único usamos el IV en el primer bloque.

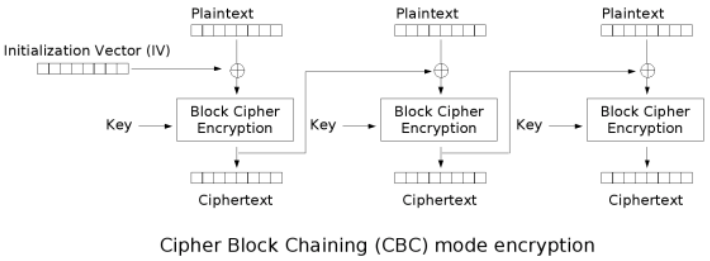

Figura 29: Secuencia del Cifrado CBC

Como el código de AES es un código muy complejo y extenso, hemos añadido dicha implementación en Lenguaje C en el Apéndice del presente documento en lugar de en el presente capitulo.

Lo que si vamos a mostrar a continuación es el fragmento de código de *simpletun* que hemos modificado para incorporar un cifrado por AES y crear nuestra VPN:

```
106 /*************************************************************************
107  * aes_enc: Ciframos con AES128-CBC el mensaje que vamos a enviar    *
108  *************************************************************************/
109 int aes_enc(char *buf,int n){
110         // ctx es una estructura que almacenará la clave y la IV expandidas
111         struct AES_ctx ctx;
112 
113         // AES.c no trabaja con char, sino con un uint8_t == unsigned char
114         uint8_t key[] = "guillemdani57000"; // clave de cifrado
115         uint8_t iv[]  = "rcotrabajotun123"; // IV (vector de inicializacion)
116         int resto, tam, res;
117 
118         // calculamos el tamaño (multiple de 16) que debe tener
119         // el buffer a cifrar
120         res = n / 16;
121         resto = n % 16;
122 
123         if(resto != 0){
124                 tam = 16 * (res+1);
125         } else { tam = n; }
126 
127         uint8_t str[tam]; // buffer con tam % 16 = 0, que cifraremos
128 
129         for (int i = 0; i < n; ++i) {
130             str[i] = (uint8_t) buf[i];  // copiamos cada char al buffer de tipo uint8_t
131         }
132 
133         AES_init_ctx_iv(&ctx, key, iv); // inicializamos la llave y IV (key_expansion)
134         AES_CBC_encrypt_buffer(&ctx, str, tam); // ciframos str con la clave
135 
136 
137         for (int i = 0; i < tam; ++i) {
138             buf[i] = (char) str[i];     // una vez cifrado, volvemos a transformar el
                                            // buffer a char
139         }
140 
141         return tam;
142 
143 }
```

Figura 30: Función de cifrado de AES128 CBC

```
/*****************************************************************************
 * aes_dec: Desciframos con AES128-CBC el mensaje que vamos a enviar    *
 *****************************************************************************/
void aes_dec(char *buf, int n, int n_aes){ // buf es el buffer con los datos cifrados
                                           // n = tamaño texto plano
                                           // n_aes = tamaño texto cifrado (n_aes % 16 = 0)
        struct AES_ctx ctx;

        uint8_t key[] = "guillemdani57000";
        uint8_t iv[]  = "rcotrabajotun123";
        uint8_t str[n_aes];

        for (int i = 0; i < n_aes; ++i) {
            str[i] = (uint8_t) buf[i];
        }

        AES_init_ctx_iv(&ctx, key, iv);
        AES_CBC_decrypt_buffer(&ctx, str, n_aes);  // desciframos pasando los mismos
                                                   // argumentos que para cifrar
                                                   // pasamos el tamaño del cifrado

        // nos quedamos solo con los n primeros uint8_t del buffer descifrado
        for (int i = 0; i < n; ++i) {
            buf[i] = (char) str[i];     // transformamos de nuevo a char el texto plano
        }
}

```

Figura31 : Función de descifrado de AES128 CBC

En nuestro código, por simplicidad, hemos decidido que tanto la clave simétrica (*key*) como la *iv* esten definidas en el código en ambas máquinas (cliente y servidor TUN). En la realidad, estas claves deberían acordarse entre ambas máquinas por medio de algun procedimiento de clave asimétrica.

Estas dos funciones, de cifrado y descifrado, deberán ser invocadas, al igual que las del cifrado César explicado anteriormente, desde el bucle de espera que se estará ejecutando en el servidor y cliente TUN. Este es el fragmento de código del bucle de espera donde se invoca a estas dos funciones AES:

```
    if(FD_ISSET(tap_fd, &rd_set)){
      /* data from tun/tap: just read it and write it to the network */

      nread = cread(tap_fd, buffer, BUFSIZE);   // nread = tamaño mensaje plano
      length_aes = aes_enc(buffer, nread);      // ciframos el mensaje plano
                                                // length_aes = tamaño mensaje cifrado

      tap2net++;
      do_debug("TAP2NET %lu: Read %d bytes from the tap interface\n", tap2net, nread);

      /* write length + packet */
      plength = htons(nread);   // formato bytes host a formato IP/TCP
        // enviamos primero tamaño
      nwrite = cwrite(net_fd, (char *)&plength, sizeof(plength));
      nwrite = cwrite(net_fd, buffer, length_aes);

      do_debug("TAP2NET %lu: Written %d bytes to the network\n", tap2net, nwrite);
    }

```

Figura 32: Rutina si llega un paquete de la interfaz TUN

```
429     if(FD_ISSET(net_fd, &rd_set)){
430       /* data from the network: read it, and write it to the tun/tap interface.
431        * We need to read the length first, and then the packet */
432
433       /* Read length */
434       nread = read_n(net_fd, (char *)&plength, sizeof(plength));
435       if(nread == 0) {
436         /* ctrl-c at the other end */
437         break;
438       }
439
440       net2tap++;
441
442       /* read packet */
443         // Primero debemos calcular el tamaño del mensaje cifrado (multiple de 16)
444         // a partir del tamaño del mensaje plano (recibido primero)
445         plength = ntohs(plength);
446         if(plength % 16 != 0){
447                 length_aes = plength / 16;
448                 length_aes = 16 * (length_aes + 1);
449         }
450
451       nread = read_n(net_fd, buffer, length_aes);       // leemos buffer
452       do_debug("NET2TAP %lu: Read %d bytes from the network\n", net2tap, nread);
453         // desciframos el mensaje cifrado
454       aes_dec(buffer, plength, length_aes);
455         // le pasamos el mensaje, su tamaño en plano y su tamaño cifrado
456
457
458       /* now buffer[] contains a full packet or frame, write it into the tun/tap interface
    */
459       nwrite = cwrite(tap_fd, buffer, nread); // lo enviamos a la interfaz TUN descifrado
460       do_debug("NET2TAP %lu: Written %d bytes to the tap interface\n", net2tap, nwrite);
461     }
462   }
```

Figura33 : Rutina si llega un paquete de la red

Debemos destacar, que AES necesita que el tamaño de los datos que va a cifrar sea múltiplo de 16, por lo que si el mensaje en plano a cifrar no lo és, se debe rellenar con zeros el resto del mensaje hasta llegar a un múltiple de 16.

Es por esto que los tamaños de los mensajes en plano y en cifrado son diferentes. Esto puede ser un incoveniente ya que cuando el emisor cifra el mensaje, el receptor cuando lo descifre no sabrá hasta que *bit* formaba parte del mensaje original. Por ello nuestra solución ha sido modificar el envío del paquete TCP en el que se indicaba el tamaño del paquete que se iba a enviar posteriormente. En vez, de enviar el tamaño del paquete cifrado, enviamos el tamaño del mensaje en plano, así el receptor sabrá hasta que *bit* tendrá que iterar tras descifrar para quedarse con el mensaje en plano. Además el receptor, calculará el tamaño del paquete cifrado (necesario para leer el *buffer*) aplicando la misma formula que aplica el emisor para calcularlo en el cifrado (Figura 33, linea 446-449).

Para comprobar el funcionamiento de nuestra VPN con cifrado AES128 realizaremos la misma prueba que realizamos con el cifrado César.

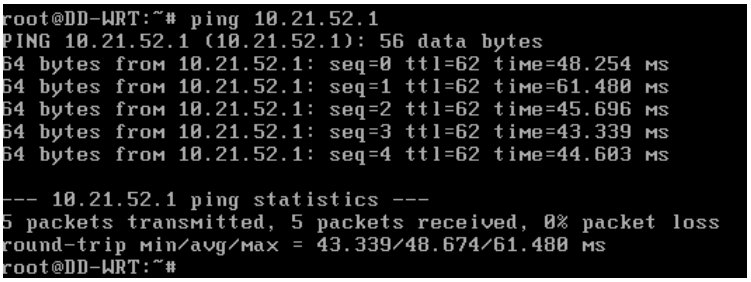

Figura 34: prueba AES - ping de DDWRT-X a DDWRT-noX

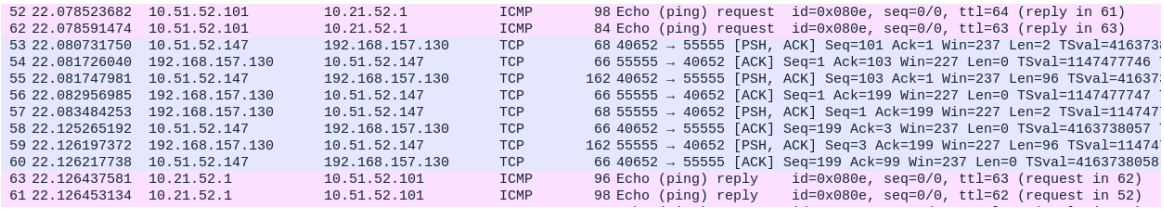

Figura 35: prueba AES - captura de los paquetes transmitidos

Vemos que el numero y secuencia de paquetes es idéntico a la prueba de César (Figura 24). Pero si que podemos ver diferencia en el tamaño del paquete TCP que lleva el ICMP Request cifrado (IP+ICMP Request), el 55 en la captura. Mientras en César era 150B (66 + 84B) aquí és de 162B (66 + 96B). Esto es por la diferencia de tamaño entre el mensaje plano y el mensaje cifrado en AES, como ya he explicado más arriba. La diferencia de 12B, son los zeros que se añaden al mensaje plano para que su tamaño sea múltiple de 16 y podamos cifrarlo con AES.

Vamos a ver los paquetes más importantes con más detalle. Los paquetes ICMP Request sin cifrar (marcados en rosa) ya hemos explicado anteriormente porque aparecen en la captura, y son idénticos a los capturados en César, salvo por el campo de datos de ICMP. Ahora simplemente mostraré uno de los dos ICMP Request con el fin de observar los datos sin cifrar y compararlos con los datos cifrados.

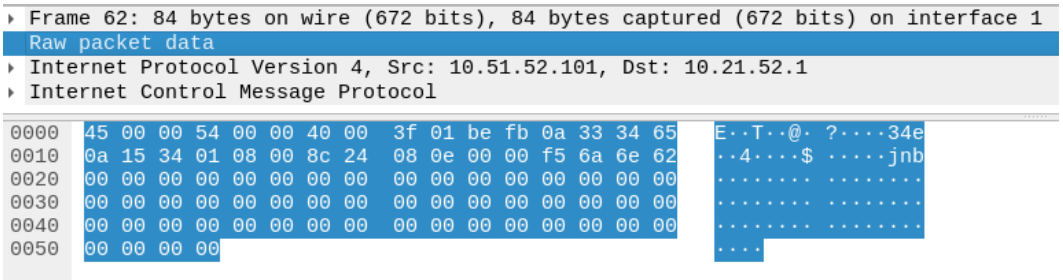

Figura 36: ICMP Request sin cifrar capturado en tun3

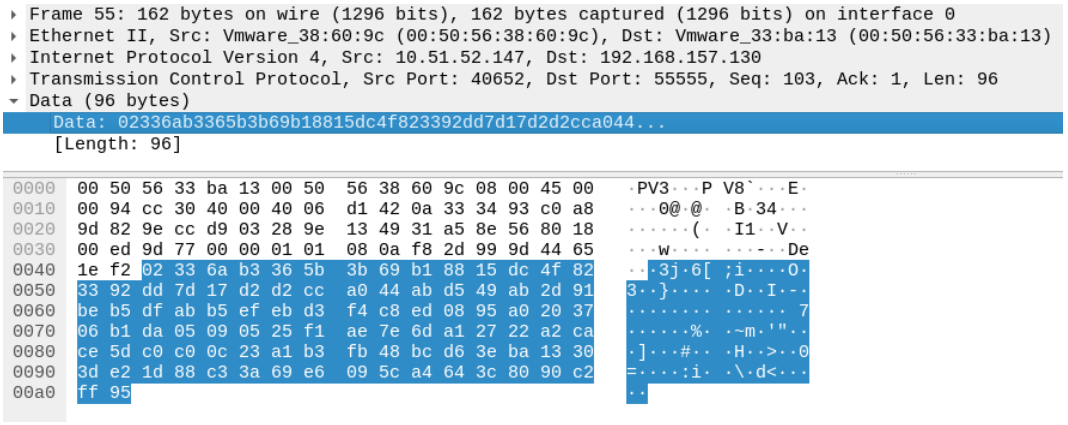

Figura37 : ICMP Request cifrado capturado en ens37

Como vemos, es imposible detectar cualquier equivalencia entre ambos datos, a diferencia del cifrado César, lo que produce que sea imposible deducir los datos originales a partir del cifrado por mucho que se intercepten paquetes.

En este ejemplo se ve perfectamente como los datos cifrados tienen un tamaño de 96B, mientras que el paquete IP sin cifrar tiene un tamaño de 84B. Todo acorde a lo que hemos explicado anteriormente.

Como hemos dicho antes, antes de enviar el cifrado, el emisor envía el tamaño del mensaje en plano.

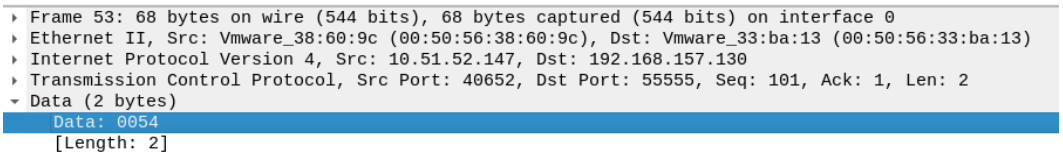

Figura 38: paquete en el que se indica el tamaño del mensaje plano

Realmente, lo más apropiado seria cifrar también el tamaño del texto original, sin embargo por simplicidad y tiempo hemos optado por dejarlo como mensaje plano.

Finalmente, si capturaramos el tráfico en el host destino del túnel, RCO-noX, podríamos ver que el paquete descifrado seria exactamente igual, tamaño incluido, al de la Figura 36.

5. Conclusión

Las VPN son una tecnología de red muy importante en la actualidad para garantizar la seguridad de nuestros sistemas informáticos. En este trabajo hemos intentado mostrar como podemos implementar una VPN sencilla a partir de la herramienta software de interfaces virtuales, TUN y de un sistema de cifrado.

TUN nos ha proporcionado una base muy interesante para crear nuestro túnel VPN, ya que gracias a utilizar un programa sencillo escrito en Lenguaje C como es *simpletun*, hemos podido modificar el código y entenderlo más a fondo. Además, la gran ventaja de usar TUN es que no es necesario usar interfaces físicas reales o virtuales, que son limitadas y más cerradas a modificarlas, y podemos utilizar ilimitadas interfaces totalmente virtuales para formar varios túneles.

En cuánto a la securización de nuestro túnel TUN, hemos aprendido el mecanismo por el cuál se cifran los datos en un túnel, creando una Red Privada Virtual (VPN). Hemos utilizado dos sistemas de cifrado totalmente opuestos, un cifrado muy sencillo y vulnerable como César, y otro más complejo y eficaz como AES128.

Por último, haciendo balance de nuestros objetivos, hemos quedado muy satisfechos de los resultados que hemos conseguido y del funcionamiento final de nuestra VPN, especialmente el cifrado AES, que pese a que ha sido una tarea compleja y que ha requerido un gran esfuerzo, hemos conseguido implementarlo en nuestro túnel TUN con resultados muy positivos.

6. Referencias

- Dworkin, M. (2021, 1 1). *Block cipher mode of operation*. Wikipedia. Retrieved November 23, 2021, from https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Cipher_Block_Chaining_.28CBC.29
- López, J. (2021, September 29). *Cifrado AES-256 bits, cómo funciona y ¿es realmente seguro?* HardZone. Retrieved November 23, 2021, from https://hardzone.es/tutoriales/rendimiento/cifrado-aes-256-bits-como-funciona/
- Mishra, N. (2021, 1 1). *Caesar Cipher in C and C++ [Encryption & Decryption]*. The Crazy Programmer. Retrieved November 22, 2021, from https://www.thecrazyprogrammer.com/2016/11/caesar-cipher-c-c-encryption-decryption.html
- Wikipedia. (2021, 1 1). *Advanced Encryption Standard*. Wikipedia. Retrieved November 23, 2021, from https://es.wikipedia.org/wiki/Advanced_Encryption_Standard
- Wikipedia. (2021, 1 1). *Cifrado César*. Wikipedia. Retrieved November 22, 2021, from https://es.wikipedia.org/wiki/Cifrado_C%C3%A9sar

7. Apéndice

AES.h

```
//#ifndef _AES_H_
#define _AES_H_

#include <stdint.h>
#include <stddef.h>

// #define the macros below to 1/0 to enable/disable the mode of operation.
//
// CBC enables AES encryption in CBC-mode of operation.
// CTR enables encryption in counter-mode.
// ECB enables the basic ECB 16-byte block algorithm. All can be enabled
simultaneously.

// The #ifndef-guard allows it to be configured before #include'ing or at compile
time.
//#ifndef CBC
  #define CBC 1
//#endif

#ifndef ECB
  #define ECB 1
#endif

#ifndef CTR
  #define CTR 1
#endif


#define AES128 1
//#define AES192 1
//#define AES256 1

#define AES_BLOCKLEN 16 // Block length in bytes - AES is 128b block only

#if defined(AES256) && (AES256 == 1)
    #define AES_KEYLEN 32
    #define AES_keyExpSize 240
#elif defined(AES192) && (AES192 == 1)
    #define AES_KEYLEN 24
    #define AES_keyExpSize 208
#else
```

```
    #define AES_KEYLEN 16   // Key length in bytes
    #define AES_keyExpSize 176
#endif

struct AES_ctx
{
  uint8_t RoundKey[AES_keyExpSize];
#if (defined(CBC) && (CBC == 1)) || (defined(CTR) && (CTR == 1))
  uint8_t Iv[AES_BLOCKLEN];
#endif
};

void AES_init_ctx(struct AES_ctx* ctx, const uint8_t* key);
#if (defined(CBC) && (CBC == 1)) || (defined(CTR) && (CTR == 1))
void AES_init_ctx_iv(struct AES_ctx* ctx, const uint8_t* key, const uint8_t* iv);
void AES_ctx_set_iv(struct AES_ctx* ctx, const uint8_t* iv);
#endif

#if defined(ECB) && (ECB == 1)
// buffer size is exactly AES_BLOCKLEN bytes;
// you need only AES_init_ctx as IV is not used in ECB
// NB: ECB is considered insecure for most uses
void AES_ECB_encrypt(const struct AES_ctx* ctx, uint8_t* buf);
void AES_ECB_decrypt(const struct AES_ctx* ctx, uint8_t* buf);

#endif // #if defined(ECB) && (ECB == !)


#if defined(CBC) && (CBC == 1)
// buffer size MUST be mutile of AES_BLOCKLEN;
// Suggest https://en.wikipedia.org/wiki/Padding_(cryptography)#PKCS7 for padding
scheme
// NOTES: you need to set IV in ctx via AES_init_ctx_iv() or AES_ctx_set_iv()
//        no IV should ever be reused with the same key
void AES_CBC_encrypt_buffer(struct AES_ctx* ctx, uint8_t* buf, size_t length);
void AES_CBC_decrypt_buffer(struct AES_ctx* ctx, uint8_t* buf, size_t length);

#endif // #if defined(CBC) && (CBC == 1)

#if defined(CTR) && (CTR == 1)

// Same function for encrypting as for decrypting.
// IV is incremented for every block, and used after encryption as XOR-compliment
for output
// Suggesting https://en.wikipedia.org/wiki/Padding_(cryptography)#PKCS7 for
padding scheme
// NOTES: you need to set IV in ctx with AES_init_ctx_iv() or AES_ctx_set_iv()
//        no IV should ever be reused with the same key
void AES_CTR_xcrypt_buffer(struct AES_ctx* ctx, uint8_t* buf, size_t length);

#endif // #if defined(CTR) && (CTR == 1)
```

```
//#endif // _AES_H_
```

AES.c

```
/*

This is an implementation of the AES algorithm, specifically ECB, CTR and CBC mode.
Block size can be chosen in aes.h - available choices are AES128, AES192, AES256.

The implementation is verified against the test vectors in:
  National Institute of Standards and Technology Special Publication 800-38A 2001 ED

ECB-AES128
----------

  plain-text:
    6bc1bee22e409f96e93d7e117393172a
    ae2d8a571e03ac9c9eb76fac45af8e51
    30c81c46a35ce411e5fbc1191a0a52ef
    f69f2445df4f9b17ad2b417be66c3710

  key:
    2b7e151628aed2a6abf7158809cf4f3c

  resulting cipher
    3ad77bb40d7a3660a89ecaf32466ef97
    f5d3d58503b9699de785895a96fdbaaf
    43b1cd7f598ece23881b00e3ed030688
    7b0c785e27e8ad3f8223207104725dd4


NOTE:   String length must be evenly divisible by 16byte (str_len % 16 == 0)
        You should pad the end of the string with zeros if this is not the case.
        For AES192/256 the key size is proportionally larger.

*/


/*****************************************************************************/
/* Includes:                                                                 */
/*****************************************************************************/
#include <string.h> // CBC mode, for memset
#include "aes.h"

/*****************************************************************************/
/* Defines:                                                                  */
/*****************************************************************************/
// The number of columns comprising a state in AES. This is a constant in AES. Value=4
#define Nb 4

#if defined(AES256) && (AES256 == 1)
    #define Nk 8
    #define Nr 14
#elif defined(AES192) && (AES192 == 1)
    #define Nk 6
    #define Nr 12
#else
    #define Nk 4        // The number of 32 bit words in a key.
    #define Nr 10       // The number of rounds in AES Cipher.
#endif

// jcallan@github points out that declaring Multiply as a function
// reduces code size considerably with the Keil ARM compiler.
// See this link for more information: https://github.com/kokke/tiny-AES-C/pull/3
#ifndef MULTIPLY_AS_A_FUNCTION
  #define MULTIPLY_AS_A_FUNCTION 0
#endif




/*****************************************************************************/
/* Private variables:                                                        */
/*****************************************************************************/
// state - array holding the intermediate results during decryption.
typedef uint8_t state_t[4][4];
```

```
// The lookup-tables are marked const so they can be placed in read-only storage instead of
RAM
// The numbers below can be computed dynamically trading ROM for RAM -
// This can be useful in (embedded) bootloader applications, where ROM is often limited.
static const uint8_t sbox[256] = {
  //0     1    2      3     4    5     6     7      8    9     A      B    C     D     E     F
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab,
0x76,
  0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72,
0xc0,
  0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31,
0x15,
  0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2,
0x75,
  0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f,
0x84,
  0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58,
0xcf,
  0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f,
0xa8,
  0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3,
0xd2,
  0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19,
0x73,
  0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b,
0xdb,
  0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4,
0x79,
  0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae,
0x08,
  0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b,
0x8a,
  0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d,
0x9e,
  0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28,
0xdf,
  0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb,
0x16 };

#if (defined(CBC) && CBC == 1) || (defined(ECB) && ECB == 1)
static const uint8_t rsbox[256] = {
  0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7,
0xfb,
  0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9,
0xcb,
  0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3,
0x4e,
  0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1,
0x25,
  0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6,
0x92,
  0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d,
0x84,
  0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45,
0x06,
  0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a,
0x6b,
  0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6,
0x73,
  0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf,
0x6e,
  0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe,
0x1b,
  0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a,
0xf4,
  0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec,
0x5f,
  0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c,
0xef,
  0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99,
0x61,
  0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c,
0x7d };
#endif
```

```
// The round constant word array, Rcon[i], contains the values given by
// x to the power (i-1) being powers of x (x is denoted as {02}) in the field GF(2^8)
static const uint8_t Rcon[11] = {
  0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36 };

/*
 * Jordan Goulder points out in PR #12 (https://github.com/kokke/tiny-AES-C/pull/12),
 * that you can remove most of the elements in the Rcon array, because they are unused.
 *
 * From Wikipedia's article on the Rijndael key schedule @
https://en.wikipedia.org/wiki/Rijndael_key_schedule#Rcon
 *
 * "Only the first some of these constants are actually used – up to rcon[10] for AES-128 (as
11 round keys are needed),
 *  up to rcon[8] for AES-192, up to rcon[7] for AES-256. rcon[0] is not used in AES
algorithm."
 */


/*****************************************************************************/
/* Private functions:                                                        */
/*****************************************************************************/
/*
static uint8_t getSBoxValue(uint8_t num)
{
  return sbox[num];
}
*/
#define getSBoxValue(num) (sbox[(num)])

// This function produces Nb(Nr+1) round keys. The round keys are used in each round to
decrypt the states.
static void KeyExpansion(uint8_t* RoundKey, const uint8_t* Key)
{
  unsigned i, j, k;
  uint8_t tempa[4]; // Used for the column/row operations

  // The first round key is the key itself.
  for (i = 0; i < Nk; ++i)
  {
    RoundKey[(i * 4) + 0] = Key[(i * 4) + 0];
    RoundKey[(i * 4) + 1] = Key[(i * 4) + 1];
    RoundKey[(i * 4) + 2] = Key[(i * 4) + 2];
    RoundKey[(i * 4) + 3] = Key[(i * 4) + 3];
  }

  // All other round keys are found from the previous round keys.
  for (i = Nk; i < Nb * (Nr + 1); ++i)
  {
    {
      k = (i - 1) * 4;
      tempa[0]=RoundKey[k + 0];
      tempa[1]=RoundKey[k + 1];
      tempa[2]=RoundKey[k + 2];
      tempa[3]=RoundKey[k + 3];

    }

    if (i % Nk == 0)
    {
      // This function shifts the 4 bytes in a word to the left once.
      // [a0,a1,a2,a3] becomes [a1,a2,a3,a0]

      // Function RotWord()
      {
        const uint8_t u8tmp = tempa[0];
        tempa[0] = tempa[1];
        tempa[1] = tempa[2];
        tempa[2] = tempa[3];
        tempa[3] = u8tmp;
      }

      // SubWord() is a function that takes a four-byte input word and
      // applies the S-box to each of the four bytes to produce an output word.

      // Function Subword()
      {
```

```
        tempa[0] = getSBoxValue(tempa[0]);
        tempa[1] = getSBoxValue(tempa[1]);
        tempa[2] = getSBoxValue(tempa[2]);
        tempa[3] = getSBoxValue(tempa[3]);
      }

      tempa[0] = tempa[0] ^ Rcon[i/Nk];
    }
#if defined(AES256) && (AES256 == 1)
    if (i % Nk == 4)
    {
      // Function Subword()
      {
        tempa[0] = getSBoxValue(tempa[0]);
        tempa[1] = getSBoxValue(tempa[1]);
        tempa[2] = getSBoxValue(tempa[2]);
        tempa[3] = getSBoxValue(tempa[3]);
      }
    }
#endif
    j = i * 4; k=(i - Nk) * 4;
    RoundKey[j + 0] = RoundKey[k + 0] ^ tempa[0];
    RoundKey[j + 1] = RoundKey[k + 1] ^ tempa[1];
    RoundKey[j + 2] = RoundKey[k + 2] ^ tempa[2];
    RoundKey[j + 3] = RoundKey[k + 3] ^ tempa[3];
  }
}

void AES_init_ctx(struct AES_ctx* ctx, const uint8_t* key)
{
  KeyExpansion(ctx->RoundKey, key);
}
#if (defined(CBC) && (CBC == 1)) || (defined(CTR) && (CTR == 1))
void AES_init_ctx_iv(struct AES_ctx* ctx, const uint8_t* key, const uint8_t* iv)
{
  KeyExpansion(ctx->RoundKey, key);
  memcpy (ctx->Iv, iv, AES_BLOCKLEN);
}
void AES_ctx_set_iv(struct AES_ctx* ctx, const uint8_t* iv)
{
  memcpy (ctx->Iv, iv, AES_BLOCKLEN);
}
#endif

// This function adds the round key to state.
// The round key is added to the state by an XOR function.
static void AddRoundKey(uint8_t round, state_t* state, const uint8_t* RoundKey)
{
  uint8_t i,j;
  for (i = 0; i < 4; ++i)
  {
    for (j = 0; j < 4; ++j)
    {
      (*state)[i][j] ^= RoundKey[(round * Nb * 4) + (i * Nb) + j];
    }
  }
}

// The SubBytes Function Substitutes the values in the
// state matrix with values in an S-box.
static void SubBytes(state_t* state)
{
  uint8_t i, j;
  for (i = 0; i < 4; ++i)
  {
    for (j = 0; j < 4; ++j)
    {
      (*state)[j][i] = getSBoxValue((*state)[j][i]);
    }
  }
}

// The ShiftRows() function shifts the rows in the state to the left.
// Each row is shifted with different offset.
// Offset = Row number. So the first row is not shifted.
static void ShiftRows(state_t* state)
{
```

```
  uint8_t temp;

  // Rotate first row 1 columns to left
  temp           = (*state)[0][1];
  (*state)[0][1] = (*state)[1][1];
  (*state)[1][1] = (*state)[2][1];
  (*state)[2][1] = (*state)[3][1];
  (*state)[3][1] = temp;

  // Rotate second row 2 columns to left
  temp           = (*state)[0][2];
  (*state)[0][2] = (*state)[2][2];
  (*state)[2][2] = temp;

  temp           = (*state)[1][2];
  (*state)[1][2] = (*state)[3][2];
  (*state)[3][2] = temp;

  // Rotate third row 3 columns to left
  temp           = (*state)[0][3];
  (*state)[0][3] = (*state)[3][3];
  (*state)[3][3] = (*state)[2][3];
  (*state)[2][3] = (*state)[1][3];
  (*state)[1][3] = temp;
}

static uint8_t xtime(uint8_t x)
{
  return ((x<<1) ^ (((x>>7) & 1) * 0x1b));
}

// MixColumns function mixes the columns of the state matrix
static void MixColumns(state_t* state)
{
  uint8_t i;
  uint8_t Tmp, Tm, t;
  for (i = 0; i < 4; ++i)
  {
    t   = (*state)[i][0];
    Tmp = (*state)[i][0] ^ (*state)[i][1] ^ (*state)[i][2] ^ (*state)[i][3] ;
    Tm  = (*state)[i][0] ^ (*state)[i][1] ; Tm = xtime(Tm);  (*state)[i][0] ^= Tm ^ Tmp ;
    Tm  = (*state)[i][1] ^ (*state)[i][2] ; Tm = xtime(Tm);  (*state)[i][1] ^= Tm ^ Tmp ;
    Tm  = (*state)[i][2] ^ (*state)[i][3] ; Tm = xtime(Tm);  (*state)[i][2] ^= Tm ^ Tmp ;
    Tm  = (*state)[i][3] ^ t ;              Tm = xtime(Tm);  (*state)[i][3] ^= Tm ^ Tmp ;
  }
}

// Multiply is used to multiply numbers in the field GF(2^8)
// Note: The last call to xtime() is unneeded, but often ends up generating a smaller binary
//       The compiler seems to be able to vectorize the operation better this way.
//       See https://github.com/kokke/tiny-AES-c/pull/34
#if MULTIPLY_AS_A_FUNCTION
static uint8_t Multiply(uint8_t x, uint8_t y)
{
  return (((y & 1) * x) ^
       ((y>>1 & 1) * xtime(x)) ^
       ((y>>2 & 1) * xtime(xtime(x))) ^
       ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^
       ((y>>4 & 1) * xtime(xtime(xtime(xtime(x))))); /* this last call to xtime() can be
omitted */
  }
#else
#define Multiply(x, y)                                \
      (  ((y & 1) * x) ^                              \
      ((y>>1 & 1) * xtime(x)) ^                       \
      ((y>>2 & 1) * xtime(xtime(x))) ^                \
      ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^         \
      ((y>>4 & 1) * xtime(xtime(xtime(xtime(x))))))   \

#endif

#if (defined(CBC) && CBC == 1) || (defined(ECB) && ECB == 1)
/*
static uint8_t getSBoxInvert(uint8_t num)
{
  return rsbox[num];
}
```

```
*/
#define getSBoxInvert(num) (rsbox[(num)])

// MixColumns function mixes the columns of the state matrix.
// The method used to multiply may be difficult to understand for the inexperienced.
// Please use the references to gain more information.
static void InvMixColumns(state_t* state)
{
  int i;
  uint8_t a, b, c, d;
  for (i = 0; i < 4; ++i)
  {
    a = (*state)[i][0];
    b = (*state)[i][1];
    c = (*state)[i][2];
    d = (*state)[i][3];

    (*state)[i][0] = Multiply(a, 0x0e) ^ Multiply(b, 0x0b) ^ Multiply(c, 0x0d) ^ Multiply(d,
0x09);
    (*state)[i][1] = Multiply(a, 0x09) ^ Multiply(b, 0x0e) ^ Multiply(c, 0x0b) ^ Multiply(d,
0x0d);
    (*state)[i][2] = Multiply(a, 0x0d) ^ Multiply(b, 0x09) ^ Multiply(c, 0x0e) ^ Multiply(d,
0x0b);
    (*state)[i][3] = Multiply(a, 0x0b) ^ Multiply(b, 0x0d) ^ Multiply(c, 0x09) ^ Multiply(d,
0x0e);
  }
}


// The SubBytes Function Substitutes the values in the
// state matrix with values in an S-box.
static void InvSubBytes(state_t* state)
{
  uint8_t i, j;
  for (i = 0; i < 4; ++i)
  {
    for (j = 0; j < 4; ++j)
    {
      (*state)[j][i] = getSBoxInvert((*state)[j][i]);
    }
  }
}

static void InvShiftRows(state_t* state)
{
  uint8_t temp;

  // Rotate first row 1 columns to right
  temp = (*state)[3][1];
  (*state)[3][1] = (*state)[2][1];
  (*state)[2][1] = (*state)[1][1];
  (*state)[1][1] = (*state)[0][1];
  (*state)[0][1] = temp;

  // Rotate second row 2 columns to right
  temp = (*state)[0][2];
  (*state)[0][2] = (*state)[2][2];
  (*state)[2][2] = temp;

  temp = (*state)[1][2];
  (*state)[1][2] = (*state)[3][2];
  (*state)[3][2] = temp;

  // Rotate third row 3 columns to right
  temp = (*state)[0][3];
  (*state)[0][3] = (*state)[1][3];
  (*state)[1][3] = (*state)[2][3];
  (*state)[2][3] = (*state)[3][3];
  (*state)[3][3] = temp;
}
#endif // #if (defined(CBC) && CBC == 1) || (defined(ECB) && ECB == 1)

// Cipher is the main function that encrypts the PlainText.
static void Cipher(state_t* state, const uint8_t* RoundKey)
{
  uint8_t round = 0;
```

```
  // Add the First round key to the state before starting the rounds.
  AddRoundKey(0, state, RoundKey);

  // There will be Nr rounds.
  // The first Nr-1 rounds are identical.
  // These Nr rounds are executed in the loop below.
  // Last one without MixColumns()
  for (round = 1; ; ++round)
  {
    SubBytes(state);
    ShiftRows(state);
    if (round == Nr) {
      break;
    }
    MixColumns(state);
    AddRoundKey(round, state, RoundKey);
  }
  // Add round key to last round
  AddRoundKey(Nr, state, RoundKey);
}

#if (defined(CBC) && CBC == 1) || (defined(ECB) && ECB == 1)
static void InvCipher(state_t* state, const uint8_t* RoundKey)
{
  uint8_t round = 0;

  // Add the First round key to the state before starting the rounds.
  AddRoundKey(Nr, state, RoundKey);

  // There will be Nr rounds.
  // The first Nr-1 rounds are identical.
  // These Nr rounds are executed in the loop below.
  // Last one without InvMixColumn()
  for (round = (Nr - 1); ; --round)
  {
    InvShiftRows(state);
    InvSubBytes(state);
    AddRoundKey(round, state, RoundKey);
    if (round == 0) {
      break;
    }
    InvMixColumns(state);
  }

}
#endif // #if (defined(CBC) && CBC == 1) || (defined(ECB) && ECB == 1)

/*****************************************************************************/
/* Public functions:                                                         */
/*****************************************************************************/
#if defined(ECB) && (ECB == 1)


void AES_ECB_encrypt(const struct AES_ctx* ctx, uint8_t* buf)
{
  // The next function call encrypts the PlainText with the Key using AES algorithm.
  Cipher((state_t*)buf, ctx->RoundKey);
}

void AES_ECB_decrypt(const struct AES_ctx* ctx, uint8_t* buf)
{
  // The next function call decrypts the PlainText with the Key using AES algorithm.
  InvCipher((state_t*)buf, ctx->RoundKey);
}


#endif // #if defined(ECB) && (ECB == 1)




#if defined(CBC) && (CBC == 1)


static void XorWithIv(uint8_t* buf, const uint8_t* Iv)
{
```

```
  uint8_t i;
  for (i = 0; i < AES_BLOCKLEN; ++i) // The block in AES is always 128bit no matter the key
size
  {
    buf[i] ^= Iv[i];
  }
}

void AES_CBC_encrypt_buffer(struct AES_ctx *ctx, uint8_t* buf, size_t length)
{
  size_t i;
  uint8_t *Iv = ctx->Iv;
  for (i = 0; i < length; i += AES_BLOCKLEN)
  {
    XorWithIv(buf, Iv);
    Cipher((state_t*)buf, ctx->RoundKey);
    Iv = buf;
    buf += AES_BLOCKLEN;
  }
  /* store Iv in ctx for next call */
  memcpy(ctx->Iv, Iv, AES_BLOCKLEN);
}

void AES_CBC_decrypt_buffer(struct AES_ctx* ctx, uint8_t* buf, size_t length)
{
  size_t i;
  uint8_t storeNextIv[AES_BLOCKLEN];
  for (i = 0; i < length; i += AES_BLOCKLEN)
  {
    memcpy(storeNextIv, buf, AES_BLOCKLEN);
    InvCipher((state_t*)buf, ctx->RoundKey);
    XorWithIv(buf, ctx->Iv);
    memcpy(ctx->Iv, storeNextIv, AES_BLOCKLEN);
    buf += AES_BLOCKLEN;
  }
}

#endif // #if defined(CBC) && (CBC == 1)



#if defined(CTR) && (CTR == 1)

/* Symmetrical operation: same function for encrypting as for decrypting. Note any IV/nonce
should never be reused with the same key */
void AES_CTR_xcrypt_buffer(struct AES_ctx* ctx, uint8_t* buf, size_t length)
{
  uint8_t buffer[AES_BLOCKLEN];

  size_t i;
  int bi;
  for (i = 0, bi = AES_BLOCKLEN; i < length; ++i, ++bi)
  {
    if (bi == AES_BLOCKLEN) /* we need to regen xor compliment in buffer */
    {
      memcpy(buffer, ctx->Iv, AES_BLOCKLEN);
      Cipher((state_t*)buffer,ctx->RoundKey);

      /* Increment Iv and handle overflow */
      for (bi = (AES_BLOCKLEN - 1); bi >= 0; --bi)
      {
  /* inc will overflow */
        if (ctx->Iv[bi] == 255)
  {
          ctx->Iv[bi] = 0;
          continue;
        }
        ctx->Iv[bi] += 1;
        break;
      }
      bi = 0;
    }

    buf[i] = (buf[i] ^ buffer[bi]);
  }
}
```

```
#endif // #if defined(CTR) && (CTR == 1)
```