VPN: The Container Version

Tyrone Novillo
Facultad de Ingeniería
Departamento de Telecomunicaciones
Universidad de Cuenca
tyrone.novillo@ucuenca.edu.ec

Sebastián Guazhima
Facultad de Ingeniería
Departamento de Telecomunicaciones
Universidad de Cuenca
sebastian.guazhima@ucuenca.edu.ec

***Resumen*—This lab explores the establishment of a Virtual Private Network (VPN) between two private networks, simulating a scenario where an organization's two sites need secure interconnection through the Internet. The setup involves the use of TUN/TAP interfaces and encapsulation to create a secure tunnel for data transmission. The topography replicates a corporate environment with two sites (Host U and Host V), each having a private network.**

The lab utilizes Python code for implementing the VPN, including tun client.py and tun server.py programs. These programs initialize TUN interfaces, set up routing tables, and demonstrate the encapsulation of IP packets for secure transmission through the VPN tunnel. The experiment also includes a tunnel-breaking scenario to analyze the behavior of the VPN connection under disruption.

Key theoretical concepts covered include VPN technologies, tunneling, encapsulation, routing in VPNs, and security considerations. The practical implementation involves configuring routing tables, managing TUN/TAP interfaces, and addressing security risks associated with VPNs.

Through this hands-on experience, the lab provides insights into the complexities of VPN deployment, emphasizing the importance of correct routing configurations, encapsulation techniques, and security measures. The abstract encapsulates the practical aspects of the lab, encompassing code development, network topologies, and theoretical principles related to VPNs.

***Index Terms*—Virtual Private Network (VPN), TUN/TAP interfaces, Encapsulation, Routing in VPNs, Python code, Network topologies**

I. Introducción

Un VPN (Virtual Private Network) se refiere a una red privada construida sobre una red pública, generalmente Internet. Los ordenadores dentro de una VPN pueden comunicarse de manera segura, como si estuvieran en una red privada real físicamente aislada del exterior, aunque su tráfico pueda pasar por una red pública. La VPN permite a los empleados acceder de manera segura a la intranet de una empresa mientras viajan; también permite a las empresas expandir sus redes privadas a lugares en todo el país y alrededor del mundo.

El objetivo de esta práctica es comprender cómo funciona una VPN. Se centra en un tipo específico de VPN (el más común), que se construye sobre la capa de transporte. Se construirá una VPN muy simple desde cero para explicar cómo funciona cada componente de la tecnología VPN. Un programa de VPN real consta de dos partes esenciales: el túnel y la encriptación. Sin embargo, este laboratorio se centra únicamente en la parte del túnel. El informe abarca los siguientes temas:

- Red Privada Virtual
- Interfaz virtual TUN/TAP
- Túneles IP
- Enrutamiento

II. Marco teórico

II-A. Virtual Private Network (VPN)

Las Redes Privadas Virtuales (VPN) son tecnologías que permiten establecer conexiones seguras y cifradas a través de redes públicas, como Internet. El objetivo principal de una VPN es proporcionar un medio seguro para la transmisión de datos a través de redes no seguras. Utiliza protocolos de cifrado y autenticación para garantizar la confidencialidad e integridad de la información transmitida. Las VPN son esenciales en situaciones donde es necesario el acceso remoto a redes privadas o la conexión segura entre diferentes ubicaciones geográficas.

Existen varios tipos de VPN, incluyendo VPN de acceso remoto, VPN de sitio a sitio y VPN de punto a punto. Cada tipo se adapta a diferentes escenarios y requisitos de conectividad. En este contexto de laboratorio, se implementa una VPN para conectar dos redes privadas a través de Internet, simulando la necesidad de una organización de interconectar sedes remotas de manera segura.

II-B. Tunneling y Encapsulación

El túnel en una VPN es una parte fundamental que permite el transporte seguro de datos a través de una red no segura. El proceso de encapsulación implica envolver los datos originales en un nuevo paquete con encabezados adicionales para asegurar la entrega segura. En el laboratorio, se aborda la tecnología de túneles TUN/TAP, donde TUN simula una interfaz de capa de red y TAP simula una interfaz de capa de enlace. Estas interfaces permiten la creación de túneles virtuales para el transporte de paquetes IP.

El encapsulamiento se realiza mediante la adición de encabezados al paquete original. En este caso, la información original de la red privada se encapsula en paquetes IP y se transmite a través del túnel. Este proceso garantiza que los datos sean protegidos durante su tránsito a través de redes no seguras, brindando una capa adicional de seguridad.

II-C. *Enrutamiento en VPN*

El enrutamiento juega un papel crucial en una VPN, ya que determina cómo se dirigen los paquetes entre las ubicaciones remotas. En este laboratorio, se aborda el desafío de configurar el enrutamiento adecuado para asegurar que los paquetes entre las dos redes privadas fluyan a través del túnel VPN. Esto implica establecer rutas específicas para el tráfico destinado a la red del otro extremo del túnel, asegurando así que los datos alcancen su destino correcto.

El enrutamiento en VPN puede implicar configuraciones complejas, especialmente en entornos empresariales con múltiples sedes y topologías de red. La correcta configuración del enrutamiento garantiza una comunicación fluida y segura entre las redes privadas interconectadas.

II-D. *Seguridad y Riesgos en VPN*

La seguridad en las VPN es esencial para proteger la confidencialidad e integridad de los datos transmitidos. Los protocolos de cifrado, como IPSec, se utilizan para garantizar la privacidad de la información. Sin embargo, también es crucial abordar los posibles riesgos y vulnerabilidades asociados con las VPN.

Uno de los riesgos comunes es la fuga de DNS, donde las consultas de resolución de nombres de dominio pueden filtrarse fuera del túnel VPN, revelando información sensible. Además, la autenticación sólida y la gestión adecuada de claves son aspectos críticos para mitigar amenazas potenciales. El marco teórico debe incluir consideraciones de seguridad para destacar la importancia de implementar prácticas seguras y estar al tanto de los riesgos asociados con las VPN.

III. Configuración de la red

Se creará un túnel VPN entre una computadora (cliente) y una puerta de enlace, permitiendo que la computadora acceda de manera segura a una red privada a través de la puerta de enlace. Se requieren al menos tres máquinas: el cliente VPN (que también sirve como Host U), el servidor VPN (el enrutador/puerta de enlace) y un host en la red privada (Host V). La configuración de la red se muestra en la Figura 1.

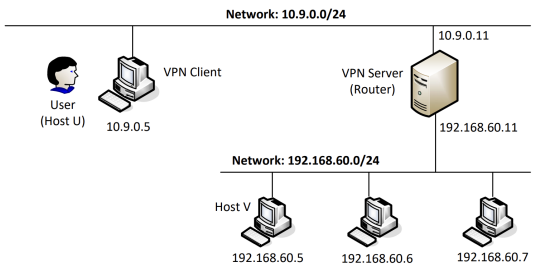

Figura 1. Topografía de la red

Los ID de los host brindados por el container son los siguientes:

```
7577ee5e1961    server-router
601abb6f97cf    client-10.9.0.5
b236c739e86d    host-192.168.60.6
ccfc9950af7f    host-192.168.60.5
```

En la práctica, el cliente VPN y el servidor VPN están conectados a través de Internet. Por simplicidad, se conectará directamente estas dos máquinas a la misma LAN, es decir, esta LAN simula Internet. La tercera máquina, Host V, es una computadora dentro de la red privada. Los usuarios en Host U (fuera de la red privada) desean comunicarse con Host V a través del túnel VPN. Para simular esta configuración, el Host V está conectado al Servidor VPN (también actuando como puerta de enlace). En tal configuración, Host V no es accesible directamente desde Internet, ni tampoco es accesible directamente desde Host U. A continuación, se demostrará que el Host U tiene la capacidad de establecer comunicación con el Servidor VPN (2), el Servidor VPN es capaz de establecer comunicación con el Host V (3), el Host U no tenga la capacidad de comunicarse con el Host V (4). Finalmente se ejecutará tcpdump en el enrutador para analizar el tráfico en cada una de las redes, demostrando así la capacidad de capturar paquetes (5). Las figuras a continuación demuestran estas configuraciones.

```
root@28c019f03563:/# ping 10.9.0.11
PING 10.9.0.11 (10.9.0.11) 56(84) bytes of data.
64 bytes from 10.9.0.11: icmp_seq=1 ttl=64 time=0.224 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.066 ms
64 bytes from 10.9.0.11: icmp_seq=3 ttl=64 time=0.187 ms
^C
--- 10.9.0.11 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2040ms
rtt min/avg/max/mdev = 0.066/0.159/0.224/0.067 ms
```

Figura 2. Comunicación entre Host U y Servidor VPN

```
root@358f55e9cbed:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=64 time=0.300 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=64 time=0.945 ms
^X^C
--- 192.168.60.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1094ms
rtt min/avg/max/mdev = 0.300/0.622/0.945/0.322 ms
```

Figura 3. Comunicación entre Servidor VPN y Host V

```
root@ccfc9950af7f:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
^C
--- 10.9.0.5 ping statistics ---
13 packets transmitted, 0 received, 100% packet loss, time 12278ms
```

Figura 4. No comunicación entre Host U y Host V

```
root@358f55e9cbed:/#*tcpdump -i eth1 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
20:53:54.509291 ARP, Request who-has 192.168.60.6 tell 192.168.60.5, length 28
20:54:29.470919 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 32, seq 1
, length 64
20:54:29.470937 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 32, seq 1,
length 64
20:54:30.494170 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 32, seq 2
, length 64
20:54:30.494314 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 32, seq 2,
length 64
20:54:34.621088 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
20:54:34.621216 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
20:54:34.621222 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
20:54:34.621223 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
^C
9 packets captured
9 packets received by filter
0 packets dropped by kernel
```

Figura 5. Sniffing de paquetes en el Servidor VPN

IV. Creación y configuración de la interfaz TUN

IV-A. Implementar un módulo de Kernel simple

La VPN a construir se basa en las tecnologías TUN/TAP. TUN y TAP son controladores de kernel de red virtual; implementan dispositivos de red que son totalmente compatibles con el software. TAP (como en network tap) simula un dispositivo Ethernet y opera con paquetes de capa 2, como tramas Ethernet; TUN (como en network TUNnel) simula un dispositivo de capa de red y opera con paquetes de capa 3, como paquetes IP. Con TUN/TAP, se puede crear interfaces de red virtuales.

Normalmente, un programa de espacio de usuario se conecta a la interfaz de red virtual TUN/TAP. Los paquetes enviados por un sistema operativo a través de una interfaz de red TUN/TAP se entregan al programa de espacio de usuario. Por otro lado, los paquetes enviados por el programa a través de una interfaz de red TUN/TAP se inyectan en la pila de red del sistema operativo.

Para el sistema operativo, parece que los paquetes provienen de una fuente externa a través de la interfaz de red virtual. Cuando un programa está conectado a una interfaz TUN/TAP, los paquetes IP enviados por el kernel a esta interfaz se dirigirán al programa. Por otro lado, los paquetes IP escritos en la interfaz por el programa se dirigirán al kernel, como si provinieran del exterior a través de esta interfaz de red virtual. El programa puede utilizar las llamadas estándar read() y write() para recibir paquetes desde o enviar paquetes a la interfaz virtual.

El script utilizado es el siguiente:

```
#!/usr/bin/env python3

import fcntl
import struct
import os
import time
from scapy.all import *

TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'tun%d', IFF_TUN |
    IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\
    x00")
print("Interface Name: {}".format(ifname))

while True:
   time.sleep(10)
```

Antes de correr el script anterior con nombre `tun.py`, se debe hacer ejecutable dicho archivo. Para ello se usan el siguiente comando en la terminal:

```
chmod a+x tun.py
```

Una vez hecho esto, se procede a ejecutar el script de python en el Host U. Una vez hecho esto, se evidencia en la figura 6 que el script ha creado una nueva interfaz llamada tun0 en el Host U.

```
root@28c019f03563:/# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: tun0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
8: eth0@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
root@28c019f03563:/#
```

Figura 6. Interfaz tun0 en el Host U

La figura 7 muestra la misma interfaz con el nombre cambiado a novillo0.

```
root@601abb6f97cf:/# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: novillo0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
16: eth0@if17: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
```

Figura 7. Interfaz novillo0 en el Host U

En este momento, la interfaz TUN no es utilizable porque aún no se ha configurado. Hay dos cosas que se debe hacer antes de que la interfaz pueda utilizarse. En primer lugar, es necesario asignarle una dirección IP. En segundo lugar, se debe activar la interfaz, ya que la interfaz todavía está en estado inactivo. Para ello se agregan los siguientes comandos para la configuración en el archivo tun.py.

```
os.system("ip addr add 192.168.53.99/24 dev {}".
    format(ifname))
os.system("ip link set dev {} up".format(ifname))
```

La figura 8 muestra el estado básico de la interfaz novillo0, a diferencia de la anterior, esta interfaz está en estado UNKNOW a diferencia de la figura 7 en la que estaba DOWN y si ndirección IP asignada.

```
root@601abb6f97cf:/volumes# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
3: novillo0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN group def
ault qlen 500
    link/none
    inet 192.168.53.99/24 scope global novillo0
       valid_lft forever preferred_lft forever
16: eth0@if17: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
```

Figura 8. Interfaz novillo0 activa y con dirección IP

Ahora se leerá desde la interfaz TUN. Todo lo que salga de la interfaz TUN es un paquete IP. Es posible convertir los datos recibidos de la interfaz en un objeto IP de Scapy, de modo que se pueda imprimir cada campo del paquete IP. Para ello, se usará el siguiente bucle while:

```
while True:
# Get a packet from the tun interface
packet = os.read(tun, 2048)
if packet:
ip = IP(packet)
print(ip.summary())
```

Al hacer ping a una red de la interfaz, se puede observar los paquetes transmitidos por dicha interfaz novillo0 (Figura 9).

```
root@601abb6f97cf:/volumes# ping 192.168.53.5 -c 2
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.
novillo0: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw
novillo0: IP / ICMP 192.168.53.99 > 192.168.53.5 echo-request 0 / Raw

--- 192.168.53.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1031ms
```

Figura 9. Lectura de paquetes enviados por la red 192.168.53.0 (Interfaz novillo0)

Al hacer un ping a un host de la red a la cual no pertenece la interfaz, los paquetes se envían pero no son recibidos por la interfaz novillo0, por lo que no pueden ser convertidos en objetos IP y po lo tanto no se muestran al momento de hacer ping (Figura 10).

```
root@601abb6f97cf:/volumes# ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1032ms
```

Figura 10. Lectura de paquetes enviados por la red 192.168.60.0

A continuación, se escribirá en la interfaz TUN. Dado que esta interfaz es una VPN, todo lo que se escriba en la interfaz por la aplicación aparecerá en el kernel como un paquete IP. Se modificará el programa tun.py, de modo que después de recibir un paquete de la interfaz TUN, un nuevo paquete será construido basado en el paquete recibido. Luego, se colocará el nuevo paquete en la interfaz TUN. Para eso, se hará uso del siguiente código de la librería Scappy de Python para hacer un Sniffing y un Spoofing de los mensajes enviados en la interfaz siempre y cuando sean paquetes ICMP echo request.

```
while True:
 # Get a packet from the tun interface
 packet = os.read(tun, 2048)
 if packet:
  pkt = IP(packet)
  print("{}:".format(ifname),pkt.summary())
  #  Task 2.d: Write to the TUN Interface
  # sniff and print out icmp echo request packet
  if ICMP in pkt and pkt[ICMP].type == 8:
    print("Original Packet.........")
    print("Source IP : ", pkt[IP].src)
    print("Destination IP :", pkt[IP].dst)
    # spoof an icmp echo reply packet
    # swap srcip and dstip
    ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=
        pkt[IP].ihl)
    icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[
        ICMP].seq)
    data = pkt[Raw].load
    newpkt = ip/icmp/data
    print("Spoofed Packet.........")
    print("Source IP : ", newpkt[IP].src)
    print("Destination IP :", newpkt[IP].dst)

    os.write(tun, bytes(newpkt))
```

V. Enviar el paquete IP al servidor VPN a través de un túnel

Se colocará el paquete IP recibido de la interfaz TUN en el campo de carga útil UDP de un nuevo paquete IP y se lo enviará a otra computadora. Es decir, se colocará el paquete original dentro de un nuevo paquete (encapsulación IP). La implementación del túnel es simplemente programación estándar cliente/servidor y puede construirse sobre TCP o UDP. En esta práctica, se usará UDP. Es decir, se va a colocar un paquete IP dentro del campo de carga útil de un paquete UDP.

Para ello se ejecutará el programa tun server.py en el servidor VPN. Este programa es simplemente un programa de servidor UDP estándar. Escucha en el puerto 9090 e imprime lo que sea que reciba. El programa asume que los datos en el campo de carga útil UDP son un paquete IP, por lo que convierte la carga útil en un objeto IP de Scapy e imprime las direcciones IP de origen y destino del paquete IP encapsulado.

El codigo ejecutado en el servidor será:

```
#!/usr/bin/env python3
from scapy.all import *
IP_A = "0.0.0.0"
PORT = 9090
sock = socket.socket(socket.AF_INET, socket.
    SOCK_DGRAM)
sock.bind((IP_A, PORT))
while True:
  data, (ip, port) = sock.recvfrom(2048)
  print("{}:{} --> {}:{}".format(ip, port, IP_A,
      PORT))
  pkt = IP(data)
  print(" Inside: {} --> {}".format(pkt.src, pkt.dst
      ))
```

El código ejecutado en el cliente será:

```
#!/usr/bin/env python3

import fcntl
import struct
import os
import time
from scapy.all import *

# Create UDP socket
sock = socket.socket(socket.AF_INET,socket.
    SOCK_DGRAM)
SERVER_IP, SERVER_PORT = "10.9.0.11", 9090

TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'novillo%d', IFF_TUN |
    IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)
```

```
# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\
    x00")
print("Interface Name: {}".format(ifname))

#Configurar la interfaz TUN
os.system("ip addr add 192.168.53.99/24 dev {}".
    format(ifname))
os.system("ip link set dev {} up".format(ifname))


while True:
  # Get a packet from the tun interface
  packet = os.read(tun, 2048)
  if packet:
    # Send the packet via the tunnel
    sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

Al hacer un ping desde el cliente hacia cualquier host dentro de la red privada de la interfaz novillo0 (192.168.53.5) usando el tunel, se observa que no existe respuesta en la parte del cliente, sin embargo, en el servidor se muestra que el tunel funciona correctamente debido a que muestra el empaquetamiento Ip en el payload UDP. Las respuestas del servidor como del cliente están en las figuras 11 y 12

```
root@601abb6f97cf:/volumes# ./tun_client.py &
[1] 23
root@601abb6f97cf:/volumes# Interface Name: novillo0

root@601abb6f97cf:/volumes# ping 192.168.53.5 -c 1
PING 192.168.53.5 (192.168.53.5) 56(84) bytes of data.

--- 192.168.53.5 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

Figura 11. Respuesta del ping 192.168.53.5 en el cliente

```
root@7577ee5e1961:/volumes# ./tun_server.py
10.9.0.5:55268 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.5
```

Figura 12. Empaquetamiento de paquetes IP sobre el payload UDP en el servidor VPN

El objetivo final es acceder a los hosts dentro de la red privada 192.168.60.0/24 utilizando el túnel. Para ello se debe configurar el enrutamiento, es decir, los paquetes dirigidos a la red 192.168.60.0/24 deben ser enrutados hacia la interfaz TUN y entregados al programa tun client.py. Se utilizará el siguiente comando para agregar una entrada a la tabla de enrutamiento:

```
ip route add 192.168.60.0/24 dev novillo0
```

Este comando especifica que los paquetes destinados a la red 192.168.60.0/24 deben ser enviados a través de la interfaz novillo0 y entregados al programa tun client.py. Es esencial seguir este procedimiento para garantizar que los paquetes de ping se envíen correctamente a través del túnel hacia la red privada, permitiendo así el acceso a los hosts dentro de la red 192.168.60.0/24.

```
10.9.0.5:55268 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
```

Figura 13. Empaquetamiento de paquetes IP sobre el payload UDP en el servidor VPN al tratar de acceder a la red LAN 192.168.60.0/24

VI. Configuración del servidor VPN

Después de que tun server.py recibe un paquete del túnel, necesita enviar el paquete al kernel para que el kernel pueda enrutar el paquete hacia su destino final. Esto debe hacerse a través de una interfaz TUN. Para ello, el servidor necesitará crear una interfaz TUN y configurarla, obtener datos desde la interfaz de socket, tratar los datos recibidos como un paquete IP y escribir el paquete en la interfaz TUN. El codigo ejecutado en el servidor será:

```
#!/usr/bin/env python3
import fcntl
import struct
import os
import time
from scapy.all import *

#TUN INTERFACE
TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'novillo%d', IFF_TUN |
    IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\
    x00")
print("Interface Name: {}".format(ifname))

#Configurar la interfaz TUN
os.system("ip addr add 192.168.53.11/24 dev {}".
    format(ifname))
os.system("ip link set dev {} up".format(ifname))


# SERVER UDP
IP_A = "0.0.0.0"
PORT = 9090
sock = socket.socket(socket.AF_INET, socket.
    SOCK_DGRAM)
sock.bind((IP_A, PORT))
while True:
  data, (ip, port) = sock.recvfrom(2048)
  print("{}:{} --> {}:{}".format(ip, port, IP_A,
      PORT))
  pkt = IP(data)
  print(" Inside: {} --> {}".format(pkt.src, pkt.dst
      ))
  os.write(tun, bytes(pkt))
```

El código ejecutado en el cliente será:

```
#!/usr/bin/env python3

import fcntl
import struct
import os
import time
from scapy.all import *
```

```
# Create UDP socket
sock = socket.socket(socket.AF_INET,socket.
    SOCK_DGRAM)
SERVER_IP, SERVER_PORT = "10.9.0.11", 9090

TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'novillo%d', IFF_TUN |
    IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\
    x00")
print("Interface Name: {}".format(ifname))

#Configurar la interfaz TUN
os.system("ip addr add 192.168.53.99/24 dev {}".
    format(ifname))
os.system("ip link set dev {} up".format(ifname))

#routing
os.system("ip route add 192.168.60.0/24 dev {}".
    format(ifname))

while True:
  # Get a packet from the tun interface
  packet = os.read(tun, 2048)
  if packet:
    # Send the packet via the tunnel
    sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

Al hacer un ping desde el Host U al Host V, los paquetes de solicitud de eco ICMP deberían llegar eventualmente al Host V a través del túnel. Es importante señalar que, aunque el Host V responderá a los paquetes ICMP, la respuesta no llegará de vuelta al Host U. Para mostrar esto, se usará tcpdump que es un sniffer de paquetes en el host V.

```
root@601abb6f97cf:/volumes# ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1028ms
```

Figura 14. Respuesta del ping 192.168.60.5 desde el Host U

```
root@7577ee5e1961:/volumes# ./tun_server.py
Interface Name: novillo0
10.9.0.5:59007 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:59007 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
```

Figura 15. Empaquetamiento de IP sobre el payload UDP desde el servidor

```
root@ccfc9950af7f:/# tcpdump -i eth0 -n 2>/dev/null
09:55:07.697644 ARP, Request who-has 192.168.60.5 tell 192.168.60.11, length 28
09:55:07.697654 ARP, Reply 192.168.60.5 is-at 02:42:c0:a8:3c:05, length 28
09:55:07.697690 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 45, seq 1
, length 64
09:55:07.697738 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 45, seq 1,
length 64
09:55:08.725958 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 45, seq 2
, length 64
09:55:08.726012 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 45, seq 2,
length 64
09:55:12.884880 ARP, Request who-has 192.168.60.11 tell 192.168.60.5, length 28
09:55:12.886087 ARP, Reply 192.168.60.11 is-at 02:42:c0:a8:3c:0b, length 28
```

Figura 16. Paquetes echo request recibidos por el host V

VII. Tráfico en ambas direcciones

Una dirección de túnel está completa, es decir, es posible enviar paquetes desde el Host U al Host V a través del túnel. Esto se debe a que el túnel es unidireccional. Para solucionar esto, se debe configurar su otra dirección para que el tráfico de retorno pueda ser enviado de vuelta al Host U.

Para ello, los programas cliente y servidor deben leer datos de dos interfaces, la interfaz TUN y la interfaz de socket. Una forma es leer de una interfaz. Por defecto, la lectura es bloqueante, es decir, el proceso se suspenderá si no hay datos. Cuando los datos estén disponibles, el proceso se desbloqueará y continuará su ejecución.

El mecanismo de bloqueo basado en lectura funciona bien para una interfaz. Si un proceso está esperando en varias interfaces, no puede bloquearse solo en una de ellas. Debe bloquearse en todas simultáneamente. Linux tiene una llamada al sistema llamada select(), que permite a un programa monitorear múltiples descriptores de archivos simultáneamente.

Para usar select(), se debe almacenar todos los descriptores de archivos que se van a monitorear en un conjunto, y luego se proporciona ese conjunto a la llamada al sistema select(), que bloqueará el proceso hasta que haya datos disponibles en uno de los descriptores de archivos en el conjunto. A continuación, se muestran respectivamente los scripts de servidor y cliente usados:

```
#!/usr/bin/env python3
import fcntl
import struct
import os
import time
from scapy.all import *

#TUN INTERFACE
TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'novillo%d', IFF_TUN |
    IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\
    x00")
print("Interface Name: {}".format(ifname))

#Configurar la interfaz TUN
os.system("ip addr add 192.168.53.11/24 dev {}".
    format(ifname))
```

```
os.system("ip link set dev {} up".format(ifname))


# SERVER UDP
IP_A = "0.0.0.0"
PORT = 9090

ip, port = '10.9.0.5', 12345

sock = socket.socket(socket.AF_INET, socket.
    SOCK_DGRAM)
sock.bind((IP_A, PORT))
while True:
# this will block until at least one interface is
    ready
  ready, _, _ = select.select([sock, tun], [], [])
  for fd in ready:
    if fd is sock:
      data, (ip, port) = sock.recvfrom(2048)
      pkt = IP(data)
      print("From socket <==: {} --> {}".format(pkt.
          src, pkt.dst))
      #... (code needs to be added by students) ...
      os.write(tun, bytes(pkt))
    if fd is tun:
      packet = os.read(tun, 2048)
      pkt = IP(packet)
      print("From tun ==>: {} --> {}".format(pkt.src
          , pkt.dst))
      sock.sendto(packet, (ip, port))
```

```
#!/usr/bin/env python3

import fcntl
import struct
import os
import time
from scapy.all import *

# Create UDP socket
sock = socket.socket(socket.AF_INET,socket.
    SOCK_DGRAM)
SERVER_IP, SERVER_PORT = "10.9.0.11", 9090

TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'novillo%d', IFF_TUN |
    IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\
    x00")
print("Interface Name: {}".format(ifname))

#Configurar la interfaz TUN
os.system("ip addr add 192.168.53.99/24 dev {}".
    format(ifname))
os.system("ip link set dev {} up".format(ifname))

#routing
os.system("ip route add 192.168.60.0/24 dev {}".
    format(ifname))

while True:
# this will block until at least one interface is
    ready
  ready, _, _ = select.select([sock, tun], [], [])
  for fd in ready:
    if fd is sock:
      data, (ip, port) = sock.recvfrom(2048)
      pkt = IP(data)
      print("From socket <==: {} --> {}".format(pkt.
          src, pkt.dst))
      #... (code needs to be added by students) ...
      os.write(tun, bytes(pkt))
    if fd is tun:
      packet = os.read(tun, 2048)
      pkt = IP(packet)
      print("From tun ==>: {} --> {}".format(pkt.src
          , pkt.dst))
      # Send the packet via the tunnel
      sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

```
root@601abb6f97cf:/volumes# ./tun_client.py &
[1] 46
root@601abb6f97cf:/volumes# Interface Name: novillo0

root@601abb6f97cf:/volumes# ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
From tun ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=2.29 ms
From tun ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=1.82 ms

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 1.815/2.054/2.294/0.239 ms
```

Figura 17. Respuesta del ping 192.168.60.5 desde el Host V hacia el Host U

```
root@601abb6f97cf:/volumes# ./tun_client.py &
[1] 46
root@601abb6f97cf:/volumes# Interface Name: novillo0

root@601abb6f97cf:/volumes# ping 192.168.60.5 -c 2
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
From tun ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=2.29 ms
From tun ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=1.82 ms

--- 192.168.60.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 1.815/2.054/2.294/0.239 ms
```

Figura 18. Empaquetamiento de IP sobre el payload UDP desde el servidor

```
10:27:42.934280 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 56, seq 1
, length 64
10:27:42.934312 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 56, seq 1,
length 64
10:27:43.936936 IP 192.168.53.99 > 192.168.60.5: ICMP echo request, id 56, seq 2
, length 64
10:27:43.936955 IP 192.168.60.5 > 192.168.53.99: ICMP echo reply, id 56, seq 2,
length 64
```

Figura 19. Paquetes echo request recibidos y enviados por el host V

Como evidencia la figura 17, la comunicación entre la Máquina V desde la Máquina U está completa, por lo tanto el túnel VPN (sin cifrar) está completo.

VIII. Experimento de Ruptura del Túnel

A continuación, se realizará una conexión Telnet desde el Host U hacia el Host V. Manteniendo la conexión telnet activa, se romperá el túnel VPN deteniendo el programa tun

server.py. Al intentar seguir con la conexión Telnet se bloquea, es decir, no permite hacer niguna acción. De igual manera sucede cuando se intenta reconectar el túnel VPN y al ejecutar los programas tun server.py,

IX. Experimento de Enrutamiento en el Host V

En un sistema VPN real, el tráfico estará cifrado. Esto significa que el tráfico de retorno debe regresar por el mismo túnel. En la configuración previa, la tabla de enrutamiento del Host V tiene una configuración predeterminada: los paquetes dirigidos a cualquier destino, excepto la red 192.168.60.0/24, se enrutarán automáticamente al servidor VPN.

En el mundo real, el Host V puede estar a varios saltos de distancia del servidor VPN, y la entrada de enrutamiento predeterminada puede no garantizar que el paquete de retorno se enrutará de vuelta al servidor VPN. Las tablas de enrutamiento dentro de una red privada deben configurarse adecuadamente para garantizar que los paquetes dirigidos al otro extremo del túnel se enrutaran hacia el servidor VPN. Para simular este escenario, se eliminará la entrada predeterminada del Host V y se agregará una entrada más específica a la tabla de enrutamiento, de modo que los paquetes de retorno puedan dirigirse de vuelta al servidor VPN.

```
ip route del default
ip route 192.168.53.0/24 via 192.168.60.11 dev eth0
```

X. VPN entre Redes Privadas

En esta tarea, se configura una VPN entre dos redes privadas. La configuración de la red se ilustra en la Figura 20.

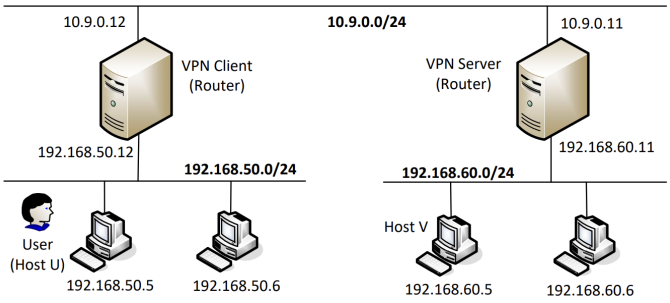

Figura 20. Topografía de la red de dos LANS conectadas mediante servidores VPN

Esta configuración simula una situación en la que una organización tiene dos sedes, cada una con una red privada. La única forma de conectar estas dos redes es a través de Internet. Para ello, establecerá una VPN entre estas dos sedes, de modo que la comunicación entre estas dos redes pase a través de un túnel VPN.

Los códigos para el cliente y el servidor fueron los siguientes:

```
#!/usr/bin/env python3

import fcntl
import struct
import os
import time
from scapy.all import *

# Create UDP socket
sock = socket.socket(socket.AF_INET,socket.
    SOCK_DGRAM)
SERVER_IP, SERVER_PORT = "10.9.0.11", 9090

TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'novillo%d', IFF_TUN |
    IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\
    x00")
print("Interface Name: {}".format(ifname))

#Configurar la interfaz TUN
os.system("ip addr add 192.168.53.99/24 dev {}".
    format(ifname))
os.system("ip link set dev {} up".format(ifname))

#routing
os.system("ip route add 192.168.60.0/24 dev {}".
    format(ifname))

while True:
# this will block until at least one interface is
    ready
  ready, _, _ = select.select([sock, tun], [], [])
  for fd in ready:
    if fd is sock:
      data, (ip, port) = sock.recvfrom(2048)
      pkt = IP(data)
      print("From socket <==: {} --> {}".format(pkt.
          src, pkt.dst))
      #... (code needs to be added by students) ...
      os.write(tun, bytes(pkt))
    if fd is tun:
      packet = os.read(tun, 2048)
      pkt = IP(packet)
      print("From tun ==>: {} --> {}".format(pkt.src
          , pkt.dst))
      # Send the packet via the tunnel
      sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

```
#!/usr/bin/env python3

import fcntl
import struct
import os
import time
from scapy.all import *

# Create UDP socket
sock = socket.socket(socket.AF_INET,socket.
    SOCK_DGRAM)
SERVER_IP, SERVER_PORT = "10.9.0.11", 9090

TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
```

```
ifr = struct.pack('16sH', b'novillo%d', IFF_TUN |
    IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\
    x00")
print("Interface Name: {}".format(ifname))

#Configurar la interfaz TUN
os.system("ip addr add 192.168.53.99/24 dev {}".
    format(ifname))
os.system("ip link set dev {} up".format(ifname))

#routing
os.system("ip route add 192.168.60.0/24 dev {}".
    format(ifname))

while True:
# this will block until at least one interface is
    ready
  ready, _, _ = select.select([sock, tun], [], [])
  for fd in ready:
    if fd is sock:
      data, (ip, port) = sock.recvfrom(2048)
      pkt = IP(data)
      print("From socket <==: {} --> {}".format(pkt.
          src, pkt.dst))
      #... (code needs to be added by students) ...
      os.write(tun, bytes(pkt))
    if fd is tun:
      packet = os.read(tun, 2048)
      pkt = IP(packet)
      print("From tun ==>: {} --> {}".format(pkt.src
          , pkt.dst))
      # Send the packet via the tunnel
      sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

Al ejecutar estos scripts, se evidenció la conexión existente entre los dos host a pesar de estar en diferentes redes.

XI. Conclusiones

La implementación de una Red Privada Virtual (VPN) proporciona una solución robusta y segura para interconectar redes privadas a través de entornos no seguros, como Internet. La utilización de tecnologías como TUN/TAP interfaces y encapsulación demuestra la versatilidad de las VPN para crear túneles seguros de comunicación, facilitando la transmisión confidencial de datos entre ubicaciones remotas. La topografía simulada de dos sitios corporativos, representados por Host U y Host V, destaca la aplicabilidad de las VPN en entornos empresariales con necesidades de conectividad remota.

Desde el punto de vista teórico, las VPNs abordan conceptos fundamentales como tunneling, encapsulación y enrutamiento. El código implementado en Python, con programas como tun client.py y tun server.py, resalta la flexibilidad y accesibilidad de las VPN mediante la automatización de procesos clave.

Referencias

[1] W. Stallings, Fundamentos de Seguridad En Redes. Pearson Publ. Co., 2004.

[2] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” Communications of the ACM, vol. 21, no. 2, pp. 120–126, 1978. DOI: 10.1145/359340.359342

[3] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, Handbook of Applied Cryptography. CRC Press, 1996.

[4] D. E. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms. Addison-Wesley Professional, 1997.

[5] W. Du, Computer & Internet Security: A hands-on approach. Independently published, 2022.

[6] C. Paar and J. Pelzl, Understanding Cryptography: A Textbook for Students and Practitioners. Springer, 2009.

[7] J. Katz and Y. Lindell, Introduction to Modern Cryptography. Chapman and Hall/CRC, 2007.

[8] R. Housley, W. Polk, W. Ford, and D. Solo, “Internet X.509 Public Key Infrastructure Certificate and CRL Profile,” RFC 2459. DOI: 10.17487/RFC2459

[9] C. Adams and S. Lloyd, Understanding PKI: Concepts, Standards, and Deployment Considerations. Addison-Wesley, 1999.