SIC Lab
Virtual Private Networks (VPNs)

version 1.3

Authors: André Zúquete, Vitor Cunha

Version log:

- 1.0: Initial version
- 1.1: Fixed references to ca_config.cfg (now to ca_extensions.cfg)
- 1.2: Removed routes from iperf host that were related with the VPN server (not necessary, are handled via the default route)
- 1.3: Reference to '/etc/openssl/client' changed to '/etc/openvpn/client'

1 Introduction

A Virtual Private Network (VPN) is an overlay networking technology that enables secure, private communication over a public network (e.g., the internet) by creating an encrypted tunnel between two endpoints (e.g., an end-device and the remote server operated by the VPN provider). This tunnel ensures that all data transmitted between the endpoints is protected from eavesdropping and tampering while in transit. If used correctly, VPNs are a fundamental tool for privacy, security, and secure remote access to private networks, services, or devices.

1.1 Incus

NOTE: We already setup Incus in Lesson 02, please skip to the "Network topology" section if you are using the same Kali VM.

Incus is a Linux Containers project that delivers a modern and secure container and virtual machine manager. It can scale easily, from a single personal computer to running a full private cloud infrastructure. The goals are to provide a unified experience for running and managing Linux-based containers or virtual machines. The project is a fork from the Canonical LXD that is maintained by some of the original LXD developers.

We will start by installing the `incus` package in Kali. You can do it using the APT package manager:

```
$ sudo apt install incus
```

Then, we need to add our current user to the incus and incus-admin groups:

```
$ sudo adduser kali incus-admin
$ newgrp incus-admin
```

After that, we must start incus and make sure it will autostart on the next boot:

```
$ sudo systemctl start incus
$ sudo systemctl enable incus
```

Now, we must initialize incus:

```
$ incus admin init --minimal
```

You should see an empty tabular output after running

```
$ incus list
+------+-------+------+------+------+-----------+
| NAME | STATE | IPV4 | IPV6 | TYPE | SNAPSHOTS |
+------+-------+------+------+------+-----------+
```

confirming the success of the incus setup.

1.2 Network topology

In this lesson we will use a similar network topology as in the firewalls guide. We will have a protected internal network which remote workers need to access via VPN. Assume the external connections arrive from the `vpn-access` network. For convenience (i.e., easy package installations) we have maintained internet access via the `incus-default` network in most containers. The iperf container will access the internet via the vpn gateway. The IP addressed in the auxiliary `incus-default` network are not relevant for this guide, as that network should not be used for anything other than package installations. In the future, if you have that need and are outside the university's firewall, you can establish VPNs over the Internet by adapting the connected networks.

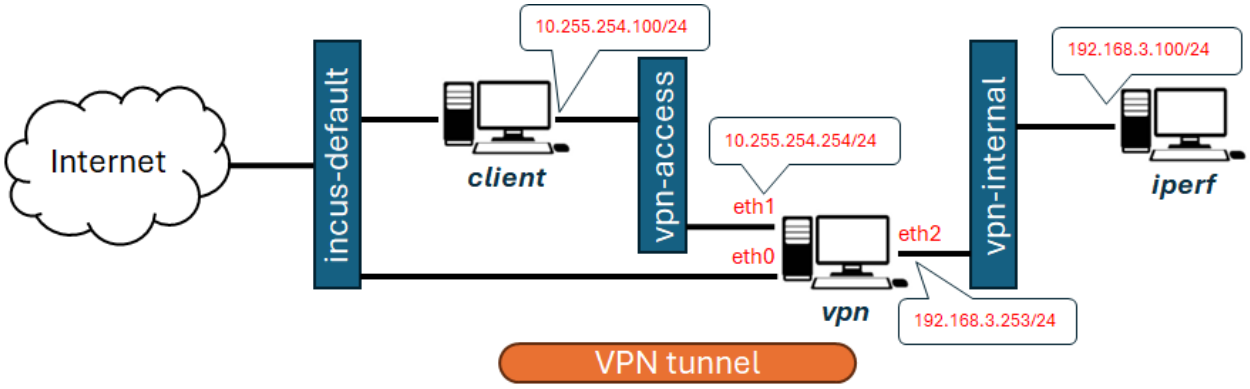

Figure 1: network topology

Create the access network:

```
$ incus network create vpn-access ipv4.address=10.255.254.1/24 ipv6.address=none ipv4.nat=false
```

This network has IP addresses managed by incus.

Create the protected network:

```
$ incus network create vpn-internal ipv4.address=none ipv6.address=none
```

This network has no IP addresses managed by incus, as these will be managed by the gateway.

And a new profile for a container that will be the gateway for our playground network:

```
$ incus profile create vpn-gw
$ incus profile device add vpn-gw eth0 nic network=incus-default name=eth0
$ incus profile device add vpn-gw eth1 nic network=vpn-access name=eth1
$ incus profile device add vpn-gw eth2 nic network=vpn-internal name=eth2
$ incus profile device add vpn-gw root disk path=/ pool=default
```

Launch the gateway:

```
$ incus launch images:debian/trixie vpn --profile vpn-gw --device eth0,ipv4.address=10.255.255.253 --device
     eth1,ipv4.address=10.255.254.254
```

Note: the second NIC (eth1) will be unconfigured, and we will deal with that later

```
$ incus list
+------+---------+----------------------+------+-----------+-----------+
| NAME |  STATE  |         IPV4         | IPV6 |   TYPE    | SNAPSHOTS |
+------+---------+----------------------+------+-----------+-----------+
| vpn  | RUNNING | 10.255.255.253 (eth0) |      | CONTAINER | 0         |
+------+---------+----------------------+------+-----------+-----------+
```

Now, because the default container image is not configured to perform DHCP on multiple interfaces, we must go within the vpn container to configure eth1 (vpn-access network). Furthermore, you need to disable the gateway from the eth1 interface.

```
$ incus shell vpn
root@vpn:~# apt install nano
root@vpn:~# nano /etc/dhcp/dhclient.conf
--> remove the `routers,` from the lintethat starts with 'request '
```

Now the interface can be configured:

```
root@vpn# dhclient -v eth1
Internet Systems Consortium DHCP Client 4.4.3-P1
Copyright 2004-2022 Internet Systems Consortium.
All rights reserved.
For info, please visit https://www.isc.org/software/dhcp/

Listening on LPF/eth1/10:66:6a:67:91:18
Sending on   LPF/eth1/10:66:6a:67:91:18
Sending on   Socket/fallback
DHCPDISCOVER on eth1 to 255.255.255.255 port 67 interval 5
DHCPOFFER of 10.255.254.254 from 10.255.254.1
DHCPREQUEST for 10.255.254.254 on eth1 to 255.255.255.255 port 67
DHCPACK of 10.255.254.254 from 10.255.254.1
bound to 10.255.254.254 -- renewal in 1429 seconds.
```

You can now check the configuration of all network interfaces, eth1 included:

```
root@vpn:~# ip a
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
    inet6 ::1/128 scope host proto kernel_lo
       valid_lft forever preferred_lft forever
29: eth0@if30: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 10:66:6a:f9:de:30 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.255.255.253/24 metric 1024 brd 10.255.255.255 scope global dynamic eth0
       valid_lft 3544sec preferred_lft 3544sec
    inet6 fe80::1266:6aff:fef9:de30/64 scope link proto kernel_ll
       valid_lft forever preferred_lft forever
31: eth1@if32: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 10:66:6a:3a:94:ef brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.255.254.254/24 brd 10.255.254.255 scope global dynamic eth1
       valid_lft 3597sec preferred_lft 3597sec
    inet6 fe80::1266:6aff:fe3a:94ef/64 scope link proto kernel_ll
       valid_lft forever preferred_lft forever
33: eth2@if34: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 10:66:6a:0a:95:dd brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::1266:6aff:fe0a:95dd/64 scope link proto kernel_ll
       valid_lft forever preferred_lft forever
```

Note: some identifiers reported by this command may be different in your setup.

The routing table should be:

```
root@vpn:~# ip r
default via 10.255.255.1 dev eth0 proto dhcp src 10.255.255.253 metric 1024
10.255.254.0/24 dev eth1 proto kernel scope link src 10.255.254.254
10.255.255.0/24 dev eth0 proto kernel scope link src 10.255.255.253 metric 1024
10.255.255.1 dev eth0 proto dhcp scope link src 10.255.255.253 metric 1024
```

And now:

```
$ incus list
+------+---------+----------------------+------+-----------+-----------+
| NAME |  STATE  |         IPV4         | IPV6 |   TYPE    | SNAPSHOTS |
+------+---------+----------------------+------+-----------+-----------+
| vpn  | RUNNING | 10.255.255.253 (eth0) |      | CONTAINER | 0         |
|      |         | 10.255.254.254 (eth1) |      |           |           |
+------+---------+----------------------+------+-----------+-----------+
```

Because the vpn-internal network addressing space is not managed by incus (by design), we will need additional configuration on the VPN gateway to: (1) activate the eth2 interface, (2) automatically provide IPv4 addresses to new services in that network, (3) perform NAT between the LAN (vpn-internal) and the WAN.

We will start by configuring eth1 by editing `/etc/systemd/network/10-eth2.network` (`vim` is already installed, install `nano` only if you need to use it):

```
$ incus shell vpn
root@vpn:~# nano /etc/systemd/network/10-eth2.network
```

And put this configuration into `/etc/systemd/network/10-eth2.network`:

```
[Match]
Name=eth2

[Network]
Address=192.168.3.253/24
```

Restart the systemd-networkd service to activate the eth2 interface with the set configuration:

```
root@vpn:~# systemctl restart systemd-networkd
```

Run `ip a` to verify your network interfaces. It should look like this:

```
root@vpn:~# ip a
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
    inet6 ::1/128 scope host proto kernel_lo
       valid_lft forever preferred_lft forever
41: eth0@if42: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 10:66:6a:08:cd:17 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.255.255.253/24 metric 1024 brd 10.255.255.255 scope global dynamic eth0
       valid_lft 3596sec preferred_lft 3596sec
    inet6 fe80::1266:6aff:fe08:cd17/64 scope link proto kernel_ll
       valid_lft forever preferred_lft forever
43: eth1@if44: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 10:66:6a:3f:df:ef brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.255.254.254/24 brd 10.255.254.255 scope global dynamic eth1
       valid_lft 3599sec preferred_lft 3599sec
    inet6 fe80::1266:6aff:fe3f:dfef/64 scope link proto kernel_ll
       valid_lft forever preferred_lft forever
45: eth2@if46: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 10:66:6a:7f:3e:0e brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.3.253/24 brd 192.168.3.255 scope global eth2
       valid_lft forever preferred_lft forever
    inet6 fe80::1266:6aff:fe7f:3e0e/64 scope link proto kernel_ll
       valid_lft forever preferred_lft forever
```

Note: some identifiers reported by this command may be different in your setup.

Now we have connectivity across the two networks (incus-default and vpn-internal). We will proceed to the installation of a DHCP server and DNS forwarder to allow the automatic configuration of new devices connected to the vpn-internal network. We will use dnsmasq to do so, a widely used utility which you can find in regular all-in-a-box Wi-Fi gateways (consumer off-the-shelf and custom solutions such as with OpenWRT, DD-WRT, Merlin, and others).

Here we will do the configuration entirely manually, to understand the controls available, and what we can subvert to exploit the local protocols.

We will start by installing `dnsmasq` and remove the systemd-resolved local DNS relay:

```
root@vpn:~# apt install dnsmasq
root@vpn:~# apt remove systemd-resolved
```

Now, edit `/etc/dnsmasq.conf` and make sure to enter the following lines:

```
interface=eth2
dhcp-range=192.168.3.100,192.168.3.150,12h
dhcp-host=iperf,192.168.3.100
```

Restart `dnsmasq` so that the configuration takes effect:

```
root@vpn:~# systemctl restart dnsmasq
```

1.2.1 Create NAT

Last but not least we will enable NAT across the two networks. The easiest solution for our purposes is to perform a plain source address replacement on the output interface that goes to the WAN. In more complex situations where granular control is not required, performing a MASQUERADE would be quicker and easier.

```
root@vpn:~# apt install iptables
root@vpn:~# iptables -t nat -A POSTROUTING ! -s 10.255.255.253 -o eth0 -j SNAT --to 10.255.255.253
```

Check if the firewall rule is properly installed:

```
root@vpn:~# iptables -nvL -t nat
Chain PREROUTING (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain POSTROUTING (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 SNAT       all  --  *      eth0   !10.255.255.253      0.0.0.0/0           to:10.255.255.253
```

1.3 Create the auxiliary containers

Let us create another profile for the containers attached only to our protected network:

```
$ incus profile create vpn-services
$ incus profile device add vpn-services eth0 nic network=vpn-internal name=eth0
$ incus profile device add vpn-services root disk path=/ pool=default
```

Let us create another profile for the access network:

```
$ incus profile create vpn-access
$ incus profile device add vpn-access eth0 nic network=incus-default name=eth0
$ incus profile device add vpn-access eth1 nic network=vpn-access name=eth1
$ incus profile device add vpn-access root disk path=/ pool=default
```

1.3.1 iperf3 server on the iperf host

Launch the iperf host:

```
$ incus launch images:debian/trixie iperf --profile vpn-services
```

We will now install iperf3. Make sure to select **yes** when asked if you want to start iperf3 automatically:

```
$ incus shell iperf
root@iperf:~# apt install iperf3
root@iperf:~# ip r
default via 192.168.3.253 dev eth0 proto dhcp src 192.168.3.100 metric 1024
192.168.3.0/24 dev eth0 proto kernel scope link src 192.168.3.100 metric 1024
192.168.3.253 dev eth0 proto dhcp scope link src 192.168.3.100 metric 1024
```

iperf3 should now be running on port TCP/5201.

1.3.2 Client

```
$ incus launch images:debian/trixie client --profile vpn-access
```

And put this configuration into `/etc/systemd/network/10-eth1.network`:

```
[Match]
Name=eth1

[Network]
Address=10.255.254.100/24
```

Restart the systemd-networkd service to activate the eth2 interface with the set configuration:

```
root@client:~# systemctl restart systemd-networkd
```

```
$ incus list
+--------+---------+----------------------+------+-----------+-----------+
|  NAME  |  STATE  |         IPV4         | IPV6 |   TYPE    | SNAPSHOTS |
+--------+---------+----------------------+------+-----------+-----------+
| client | RUNNING | 10.255.255.225 (eth0) |      | CONTAINER | 0         |
|        |         | 10.255.254.100 (eth1) |      |           |           |
+--------+---------+----------------------+------+-----------+-----------+
| iperf  | RUNNING | 192.168.3.100 (eth0)  |      | CONTAINER | 0         |
+--------+---------+----------------------+------+-----------+-----------+
| vpn    | RUNNING | 192.168.3.253 (eth2)  |      | CONTAINER | 0         |
|        |         | 10.255.255.253 (eth0) |      |           |           |
|        |         | 10.255.254.254 (eth1) |      |           |           |
+--------+---------+----------------------+------+-----------+-----------+
```

Note: the client IP address for eth0 may be different in your setup.

We will now install iperf on the client. You may select **no** when asked if you want to start iperf3 automatically:

```
$ incus shell client
root@client:~# apt install iperf3
```

1.4 Establish a baseline

```
root@client:~# ip r add 192.168.3.0/24 via 10.255.254.254
root@client:~# ping 192.168.3.100
PING 192.168.3.100 (192.168.3.100) 56(84) bytes of data.
64 bytes from 192.168.3.100: icmp_seq=1 ttl=63 time=0.143 ms
^C
```

Single connection:

```
root@client:~# iperf3 -c 192.168.3.100
Connecting to host 192.168.3.100, port 5201
[  5] local 10.255.254.100 port 50266 connected to 192.168.3.100 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  1.73 GBytes  14.8 Gbits/sec  398    631 KBytes
[  5]   1.00-2.00   sec  2.55 GBytes  21.9 Gbits/sec    2    740 KBytes
[  5]   2.00-3.00   sec  2.07 GBytes  17.8 Gbits/sec  331    773 KBytes
[  5]   3.00-4.00   sec  1.12 GBytes  9.61 Gbits/sec    1   1010 KBytes
[  5]   4.00-5.00   sec   656 MBytes  5.51 Gbits/sec    0   1.04 MBytes
[  5]   5.00-6.00   sec  1.86 GBytes  15.9 Gbits/sec  444   1.19 MBytes
[  5]   6.00-7.00   sec  1.53 GBytes  13.2 Gbits/sec    1    919 KBytes
[  5]   7.00-8.01   sec  1.69 GBytes  14.5 Gbits/sec  451   1.10 MBytes
[  5]   8.01-9.00   sec  1.83 GBytes  15.8 Gbits/sec  348   1.21 MBytes
[  5]   9.00-10.00  sec  2.51 GBytes  21.4 Gbits/sec    0   1.27 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  17.6 GBytes  15.1 Gbits/sec  1976            sender
[  5]   0.00-10.01  sec  17.6 GBytes  15.1 Gbits/sec                  receiver

iperf Done.
```

Parallel connections = 5

```
root@client:~# iperf3 -c 192.168.3.100 -P 5
Connecting to host 192.168.3.100, port 5201
(...)
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  6.86 GBytes  5.89 Gbits/sec  12855             sender
[  5]   0.00-10.00  sec  6.86 GBytes  5.89 Gbits/sec                  receiver
[  7]   0.00-10.00  sec  8.20 GBytes  7.04 Gbits/sec  12066             sender
[  7]   0.00-10.00  sec  8.20 GBytes  7.04 Gbits/sec                  receiver
[  9]   0.00-10.00  sec  9.25 GBytes  7.94 Gbits/sec  315              sender
[  9]   0.00-10.00  sec  9.25 GBytes  7.94 Gbits/sec                  receiver
[ 11]   0.00-10.00  sec  7.84 GBytes  6.73 Gbits/sec  1820             sender
[ 11]   0.00-10.00  sec  7.84 GBytes  6.73 Gbits/sec                  receiver
[ 13]   0.00-10.00  sec  8.90 GBytes  7.64 Gbits/sec  5083             sender
[ 13]   0.00-10.00  sec  8.90 GBytes  7.64 Gbits/sec                  receiver
[SUM]   0.00-10.00  sec  41.1 GBytes  35.2 Gbits/sec  32139             sender
[SUM]   0.00-10.00  sec  41.1 GBytes  35.2 Gbits/sec                  receiver
```

Now remove the route:

```
root@client:~# ip r del 192.168.3.0/24 via 10.255.254.254
```

1.5 OpenVPN

OpenVPN uses a custom TLS approach for authentication and achieving an encrypted tunnel, plus additional control messages to improve the tunnel reliability and, consequently, performance.

OpenVPN exists in two flavors, a commercial version and a community edition. In this guide we will explore the OpenVPN community edition. This edition is already pre-packaged in most Linux distributions.

Install the OpenVPN server on the vpn-gw container:

```
root@vpn:~# apt install --no-install-recommends openvpn
```

The package comes with an example of configuration file for the VPN server. This is the file 'server.conf' in the directory '/usr/share/doc/openvpn/examples/sample-config-files'. This file was already edited and you can find it in the class resources ('server.conf'). The OpenVPN server configuration file must have that name ('server.conf') and exist in the directory '/etc/openvpn'.

This configuration assumes that we have a set of cryptographic key pairs to authenticate the participants (i.e., servers and clients) with access to the VPN within in your organization. The public component of these key pairs needs to be certified, and for that purpose we will create our own, private Certification Authority (CA) to issue the public key certificates for our clients and servers.

We will use openssl commands to perform all required actions involving asymmetric key pairs:

- Key pair generation;
- Certificate Signing Request (CSR) generation; and
- Certificate issuing processes.

```
root@vpn:~# apt install openssl
```

Remember to set the adequate **basic constraints** for each certificate, the CA should have the **CA property**, the servers should have the **TLS Web Server Authentication property**, and the clients should have the **TLS Web Client Authentication** property. It is a good practice to put the hostname (i.e., fully qualified domain name – FQDN) in the common name (CN) of the server certificates. In our case, because we do not have a FQDN, simply use the container name.

The CA's private key should not be distributed under any circumstance. The VPN server private key (server.pem) should only be used in the vpn gateway. All certificates are public.

Edit a 'ca_extensions.cnf' file, and add to it the following content:

```
[Root_CA]
basicConstraints = critical, CA:TRUE
keyUsage = critical, keyCertSign

[VPN_Server]
basicConstraints = critical, CA:FALSE
keyUsage = critical, digitalSignature, keyEncipherment, dataEncipherment, keyAgreement
extendedKeyUsage = critical, serverAuth

[VPN_Client]
basicConstraints = critical, CA:FALSE
keyUsage = critical, digitalSignature, keyEncipherment, dataEncipherment, keyAgreement
extendedKeyUsage = critical, clientAuth
```

First, generate the CA's key pair (we will use EC cryptography, namely P-256) and we will not protect the private key with any password:

```
root@vpn:~# openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-384 -out ca_keys.pem
```

The file 'ca_keys.pem' will contain the CA's key pair. Now, we will create a CSR (Certificate Signing Request) from the CA's public key.

```
root@vpn:~# openssl req -new -key ca_keys.pem -out ca_csr.pem -subj "/CN=Root CA"
```

Once having our CSR, we will issue a self-certified certificate for the CA:

```
root@vpn:~# openssl x509 -signkey ca_keys.pem -in ca_csr.pem -req -days 365 -out ca_crt.pem -extensions Root_CA -extfile
     ca_extensions.cnf
```

Now, repeat the same sequence of openssl commands to generate a key pair, build a CSR and issue a certificate for the OpenVPN server:

```
root@vpn:~# openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-384 -out server_keys.pem
root@vpn:~# openssl req -new -key server_keys.pem -out server_csr.pem -subj "/CN=VPN server"
```

but for issuing an end-user certificate we need to use a slightly different command. We need to use the CA certificate to get its name, the private key is not enough:

```
root@vpn:~# openssl x509 -CA ca_crt.pem -CAkey ca_keys.pem -CAcreateserial -in server_csr.pem -req -days 365 -out
     server_crt.pem -extensions VPN_Server -extfile ca_extensions.cnf
```

Copy the files 'ca_cert.pem', 'server_crt.pem' and 'server_keys.pem' to `/etc/openvpn/server/`. The OpenVPN configuration file already expects them.

Now, the OpenVPN server uses the Diffie-Hellman key distribution protocol. But the usage of this protocol requires the setup of a set of common values, called Diffie-Hellman group (a group generator of a modular multiplicative group). In practice, the OpenVPN server needs to produce and publish the too values, g and p, that will be used in all the Diffie-Hellman key agreements with it.

We will use openssl for that, indicating a prime modulus with 2048 bits:

```
openssl dhparam -out /etc/openvpn/server/dh2048.pem 2048
```

The OpenVPN server configuration file already expects this file name, 'dh2048.pem'.

Finally, OpenVPN includes an HMAC-based authentication mechanism for filtering-out unwanted clients. This mechanism uses a group key, called the tls-auth key, that should exist in the server and in all authorized clients.

We are going to enable HMAC authentication in our server, so you need to create a 'ta.key' and distribute it to clients and servers alike:

```
openvpn --genkey tls-auth /etc/openvpn/server/ta.key
```

The OpenVPN server configuration file already expects this file name, 'ta.key'.

Start the server:

```
root@vpn:~# systemctl enable openvpn@server
Created symlink '/etc/systemd/system/multi-user.target.wants/openvpn@server.service' ->
     '/usr/lib/systemd/system/openvpn@.service'.
root@vpn:~# systemctl start openvpn@server
```

Use the same command to install the OpenVPN client and the openssl tool on the client container:

```
root@client:~# apt install --no-install-recommends openvpn
root@client:~# apt install openssl
```

The package comes with an example of configuration file for the VPN client. This is the file 'client.conf' in the directory '/usr/share/doc/openvpn/examples/sample-config-files'. This file was already edited and you can find it in the class resources ('client.conf'). The OpenVPN client configuration file must have that name ('client.conf') and exist in the directory '/etc/openvpn'.

Now we need to generate a key pair for the client, and a certificate for its public key:

```
root@client:~# openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-384 -out client_keys.pem
root@client:~# openssl req -new -key client_keys.pem -out client_csr.pem -subj "/CN=VPN client"
```

Copy the contents of 'client_csr.pem' to the vpn host, and use it in that host to issue the client certificate:

```
root@vpn:~# openssl x509 -CA ca_crt.pem -CAkey ca_keys.pem -CAcreateserial -in client_csr.pem -req -days 365 -out
     client_crt.pem -extensions VPN_Client -extfile ca_extensions.cnf
```

Copy the certificate in 'client_crt.pem' produced in the vpn host to an homonymous file in the directory '/etc/openvpn/client' of the client host. Do the same with the files 'ca_crt.pem' and 'ta.key'.

Start the server:

```
root@client:~# systemctl enable openvpn@client
Created symlink '/etc/systemd/system/multi-user.target.wants/openvpn@client.service' ->
     '/usr/lib/systemd/system/openvpn@.service'.
root@vpn:~# systemctl start openvpn@client
```

Now, you already have a VPN between the client host and the VPN host. And you are using L3 routing, which means that the VPN server will behave as a gateway. This means that TCP/IPs addresses from the client network will mapped into **tun0** TCP/IP address in the vpn host (10.8.0.0/24).

```
root@vpn:~# ip a
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
    inet6 ::1/128 scope host proto kernel_lo
       valid_lft forever preferred_lft forever
3: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN group default qlen 500
    link/none
    inet 10.8.0.1/24 scope global tun0
       valid_lft forever preferred_lft forever
    inet6 fe80::2354:2a38:2386:c8a1/64 scope link stable-privacy proto kernel_ll
       valid_lft forever preferred_lft forever
6: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 10:66:6a:8a:26:4d brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.255.255.253/24 metric 1024 brd 10.255.255.255 scope global dynamic eth0
       valid_lft 2733sec preferred_lft 2733sec
    inet6 fe80::1266:6aff:fe8a:264d/64 scope link proto kernel_ll
       valid_lft forever preferred_lft forever
8: eth1@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 10:66:6a:67:91:18 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.255.254.254/24 brd 10.255.254.255 scope global dynamic eth1
       valid_lft 2792sec preferred_lft 2792sec
    inet6 fe80::1266:6aff:fe67:9118/64 scope link proto kernel_ll
       valid_lft forever preferred_lft forever
10: eth2@if11: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 10:66:6a:fa:8b:6d brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.3.253/24 brd 192.168.3.255 scope global eth2
       valid_lft forever preferred_lft forever
    inet6 fe80::1266:6aff:fefa:8b6d/64 scope link proto kernel_ll
       valid_lft forever preferred_lft forever
```

However, the client still does not know how to reach the iperf host, because it has no route to its network (192.168.3.00/24).

This can be added by hand in the client, or automatically by the VPN server. Edit the 'server.conf' file, uncomment and correct the line that uses the push command, save the file, and restart the VPN server:

```
root@vpn:~# systemctl restart openvpn@server
```

Now do the same in the client:

```
root@client:~# systemctl restart openvpn@client
```

Check the IP routes, you now have a route for the client network that goes through the **tun0** device (the VPN client entry point):

```
root@client:~# ip r
default via 10.255.255.1 dev eth0 proto dhcp src 10.255.255.43 metric 1024
```

```
10.8.0.0/24 dev tun0 proto kernel scope link src 10.8.0.2
10.255.254.0/24 dev eth1 proto kernel scope link src 10.255.254.100
10.255.255.0/24 dev eth0 proto kernel scope link src 10.255.255.43 metric 1024
10.255.255.1 dev eth0 proto dhcp scope link src 10.255.255.43 metric 1024
192.168.3.0/24 via 10.8.0.1 dev tun0
```

Now you can ping the iperf host from the client or uses iperf3:

```
oot@client:~# ping 192.168.3.100
PING 192.168.3.100 (192.168.3.100) 56(84) bytes of data.
64 bytes from 192.168.3.100: icmp_seq=1 ttl=63 time=0.569 ms
64 bytes from 192.168.3.100: icmp_seq=2 ttl=63 time=0.493 ms
64 bytes from 192.168.3.100: icmp_seq=3 ttl=63 time=0.417 ms
^C
--- 192.168.3.100 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2050ms
rtt min/avg/max/mdev = 0.417/0.493/0.569/0.062 ms
root@client:~# iperf3 -c 192.168.3.100
Connecting to host 192.168.3.100, port 5201
[  5] local 10.8.0.2 port 53156 connected to 192.168.3.100 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  84.0 MBytes   704 Mbits/sec  167    183 KBytes
[  5]   1.00-2.00   sec  85.6 MBytes   718 Mbits/sec  101    125 KBytes
[  5]   2.00-3.00   sec  70.5 MBytes   591 Mbits/sec   87    163 KBytes
[  5]   3.00-4.00   sec  67.5 MBytes   566 Mbits/sec   78    141 KBytes
[  5]   4.00-5.00   sec  79.4 MBytes   666 Mbits/sec   61    174 KBytes
[  5]   5.00-6.00   sec  75.5 MBytes   633 Mbits/sec   83    186 KBytes
[  5]   6.00-7.00   sec  76.9 MBytes   645 Mbits/sec   92    211 KBytes
[  5]   7.00-8.00   sec  76.8 MBytes   644 Mbits/sec  104    163 KBytes
[  5]   8.00-9.00   sec  80.4 MBytes   674 Mbits/sec  124    145 KBytes
[  5]   9.00-10.00  sec  78.8 MBytes   660 Mbits/sec   33    138 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec   775 MBytes   650 Mbits/sec  930            sender
[  5]   0.00-10.00  sec   775 MBytes   650 Mbits/sec                  receiver

iperf Done.
```

You can clearly see that the performance dropped dramatically when comparing with the communication without the VPN. Unfortunately, that is a normal consequence of using cryptographic protection methods.

You can check how different data-ciphers, TLS-suites, and hash function for the authentication affect the performance.

Hint: you need to change those parameters in both the server and client.

1.6 WireGuard

Make sure OpenVPN is stopped -or- remove the route to the vpn-internal network (192.168.3.0/24).

Install the wireguard on the vpn-gw container:

```
root@vpn:~# apt install --no-install-recommends wireguard
```

Install the wireguard on the client container:

```
root@client:~# apt install --no-install-recommends wireguard
```

Generate the server key pair. The files `privatekey` and `publickey` hold the respective keys of the keypair.

```
root@vpn:~# wg genkey > privatekey
root@vpn:~# wg pubkey < privatekey > publickey
```

Now, you need to generate the client keypair:

```
root@client:~# wg genkey > privatekey
root@client:~# wg pubkey < privatekey > publickey
```

Consider the following /etc/wireguard/wg-server.conf:

```
[Interface]
Address = 10.9.0.1/24
MTU = 1420
ListenPort = 51820
PrivateKey = (server_privatekey)

[Peer]
PublicKey = (client_publickey)
AllowedIPs = 10.9.0.2/32
```

Start the server:

```
root@vpn:~# systemctl enable wg-quick@wg-server
Created symlink '/etc/systemd/system/multi-user.target.wants/wg-quick@wg-server.service' ->
     '/usr/lib/systemd/system/wg-quick@.service'.
root@vpn:~# systemctl start wg-quick@wg-server
```

And the following /etc/wireguard/wg-client.conf:

```
[Interface]
PrivateKey = (client_priv)
Address = 10.9.0.2/24
MTU = 1390

[Peer]
PublicKey = (server_pub)
Endpoint = 10.255.254.254:51820
```

Start the client:

```
root@vpn:~# systemctl enable wg-quick@wg-client
Created symlink '/etc/systemd/system/multi-user.target.wants/wg-quick@wg-client.service' ->
     '/usr/lib/systemd/system/wg-quick@.service'.
root@vpn:~# systemctl start wg-quick@wg-client
```

–> Check for basic connectivity to 10.9.0.1.

–> In wg-client.conf create a route to the vpn-internal network (192.168.3.0/24).

Consider the `AllowedIPs = x.x.x.x/x` option in the [Peer] section.

–> Use iperf3 to benchmark the performance.

2 Final words

After configuring and evaluating both OpenVPN and Wireguard, which one was more performant in your machine, and under which configuration?

Despite this guide still using the client-server mindset, WireGuard actually works more in a Peer-to-Peer mindset. What are the design implications of this choice? (e.g., consider NATed devices and potential workarounds).

WireGuard can be very powerful and quick to configure when connecting a few peers. However, the lack of a CA and need for key distribution on a per-peer basis makes it scale quite poorly. To solve this issue, VPN controllers automate the process and allow for a single point of control to reconfigure the entire P2P "mesh" network. Read more about services such as Tailscale and the self-hosted open-source alternative Headscale.

Furthermore, OpenVPN with its custom TLS approach and WireGuard with its custom protocol are easier to detect with today's Deep Packet Inspection (DPI) firewalls. You may need an obfuscation proxy such as obfsproxy or v2ray to hide those patterns. However, most restrictive firewalls will only work with known protocols, therefore using a fully compliant TLS (as in HTTPS) tunnel will be more effective. Please consider

reading more about OpenConnect VPN Server for those scenarios. These types of tunnels are already used in many corporate environments under vendor brands such as Cisco AnyConnect, Fortinet Fortigate SSL VPN, or Juniper SSL VPN (just to name a few), and these approaches are known to bypass nation-wide firewalls in restrictive countries.

3 Clean Up

After completing the guide you may want to remove the containers and save some resources. To do so:

```
$ incus stop vpn
$ incus stop iperf
$ incus stop client
$ incus remove vpn
$ incus remove iperf
$ incus remove client
$ incus profile delete vpn-access
$ incus profile delete vpn-services
$ incus profile delete vpn-gw
$ incus network delete vpn-access
$ incus network delete vpn-internal
```

4 Acknowledgements

Authored by André Zúquete, and Vítor Cunha

5 Bibliography

- OpenVPN
- WireGuard
- Tailscale
- Headscale
- obfsproxy
- v2ray
- OpenConnect VPN Server