Secure communication channel configurations
Validation with a namespace-based infrastructure

Introduction

This guide shows how to emulate a **two-site VPN** by combining **WSL2**, Linux **network namespaces**, and **StrongSwan**, a state-of-the-art implementation for channel protection. The purpose is to *validate secure-channel configurations* by enforcing them in a simplified infrastructure and observing what happens "on the wire". Furthermore, some real-world WAN impairments and characteristics are added to bring the infrastructure closer to a concrete **cloud** scenario.

- **Namespaces.** Four network namespaces are instantiated:
  - *hA*, *hB*: two ordinary hosts *10.0.1.2* (Host-A) and *10.0.2.2* (Host-B) in two subnets *10.0.1.0/24* (LAN-A) and *10.0.2.0/24* (LAN-B);
  - *gwA*, *gwB*: two VPN gateways, each with a LAN-side interface *10.0.1.1* (GW-A) and *10.0.2.1* (GW-B), and a WAN-side interface *192.0.2.1* (GW-A) and *192.0.2.2* (GW-B)
- **Virtual cabling.** Three veth pairs complete the physical layout entirely in RAM:

Host-A (hA) ↔ GW-A (gwA) *GW-A (gwA) ↔ GW-B (gwB)* *GW-B (gwB) ↔ Host-B (hB)*

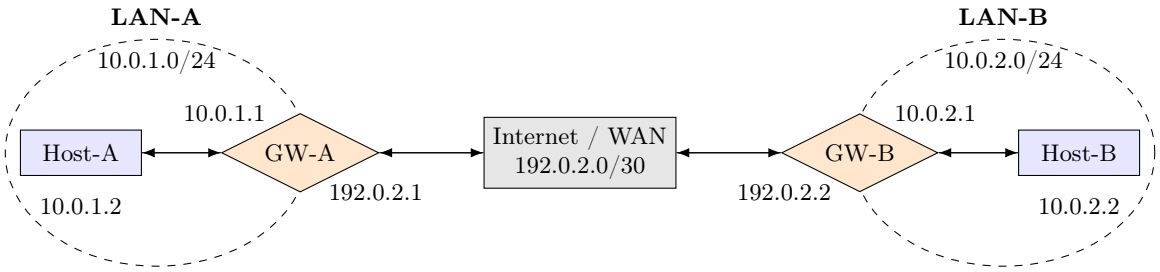

Figure 1: High-level network topology

0 Prerequisites (one-time)

WSL2 is required, and it is suggested to have the most recent stable version of the kernel. Verify at `https://github.com/microsoft/WSL/releases` for further information.

```
# check WSL/kernel in Windows shell
wsl --version    # should print "version 2.x.x.x"
```

Check and eventually install every tool used later in the guide: *iproute2* for namespaces, *tcpdump* for packet checks, *iperf3* for throughput tests, and the *StrongSwan* infrastructure for secure channel enforcement and validation.

```
# WSL shell
sudo apt update
sudo apt install -y iproute2 tcpdump iperf3 strongswan strongswan-swanctl
    strongswan-pki
```

Setting *systemd=true* converts the WSL distro from a minimal container-style environment into one that behaves like a normal Linux host with *PID=1*, which is what daemons such as *charon-systemd* (i.e., strongSwan's daemon) expect. Edit the following file as described:
`/etc/wsl.conf`

```
[boot]
systemd=true
```

Shutdown WSL2 and restart it to confirm the configuration.

```
# Reboot the VM in Windows shell
wsl --shutdown
# Back in WSL shell
ps -p 1 -o comm=    # should print "systemd"
```

1 Namespaces setup

1.0 Target topology

Four namespaces simulate two LAN hosts and two IP-forwarding gateways connected by a /30 WAN.

```
# Basic topology, not shell commands
LAN-A host          10.0.1.2/24     (ns: hA,  vAHost)
Gateway-A LAN       10.0.1.1/24     (ns: gwA, vAGwLan)
Gateway-A WAN       192.0.2.1/30    (ns: gwA, vAGwWan)
Gateway-B WAN       192.0.2.2/30    (ns: gwB, vBGwWan)
Gateway-B LAN       10.0.2.1/24     (ns: gwB, vBGwLan)
LAN-B host          10.0.2.2/24     (ns: hB,  vBHost)
```

1.1 Create the namespaces

Namespaces isolate interfaces, routes, and firewall rules so each element acts like an independent machine.

```
# Acquire sudo privileges
sudo -s
# Create namespaces
for ns in hA gwA gwB hB;
    do
        ip netns add $ns;
    done
# Verify
ip netns list   # should print "hB gwB gwA hA"
```

1.2 Create three veth cables

Each veth pair is a zero-latency Ethernet cable inside the RAM. Three are needed: LAN-A ↔ GW-A, GW-A ↔ GW-B, LAN-B ↔ GW-B.

```
# Create the veth pairs
ip link add vAHost type veth peer name vAGwLan    # LAN-A <-> GW-A
ip link add vAGwWan type veth peer name vBGwWan    # GW-A <-> GW-B (WAN)
ip link add vBGwLan type veth peer name vBHost    # GW-B <-> LAN-B
# Verify the pairs
ip link show type veth   # should print the three pairs in both directions
```

1.3 Move veth ends into their namespaces

Interfaces start life in the default (root) namespace. It is necessary to move each end to the proper namespace and attach it to the correct host.

```
# Attach the ends of the veth cables
ip link set vAHost netns hA
ip link set vAGwLan netns gwA
ip link set vAGwWan netns gwA
ip link set vBGwWan netns gwB
ip link set vBGwLan netns gwB
ip link set vBHost netns hB
# Visibility checks, should all print each interface's details
ip -n hA link show vAHost
ip -n gwA link show vAGwLan
ip -n gwA link show vAGwWan
ip -n gwB link show vBGwWan
ip -n gwB link show vBGwLan
ip -n hB link show vBHost
```

1.4 Bring up interfaces and assign IPs

Linux leaves new links DOWN, and routing fails if that is left as is.

```
# Bring up loopbacks
for ns in hA gwA gwB hB;
    do
        ip -n $ns link set lo up;
    done
```

IP addresses must be assigned to all the devices according to the basic network topology. Moreover, the states of the interfaces must be set to *UP* and the default routes for the host must be specified.

```
# LAN-A
ip -n hA addr add 10.0.1.2/24 dev vAHost
ip -n hA link set vAHost up
ip -n hA route add default via 10.0.1.1
# Gateway-A
ip -n gwA addr add 10.0.1.1/24 dev vAGwLan
ip -n gwA addr add 192.0.2.1/30 dev vAGwWan
ip -n gwA link set vAGwLan up
ip -n gwA link set vAGwWan up
# Gateway-B
ip -n gwB addr add 192.0.2.2/30 dev vBGwWan
ip -n gwB addr add 10.0.2.1/24 dev vBGwLan
ip -n gwB link set vBGwWan up
ip -n gwB link set vBGwLan up
# LAN-B
ip -n hB addr add 10.0.2.2/24 dev vBHost
ip -n hB link set vBHost up
ip -n hB route add default via 10.0.2.1
```

The addresses and state for the interfaces of all namespaces have been added and configured. They should all be in *UP* state.

```
ip -n hA -br addr show    # should print its one interface
ip -n gwA -br addr show    # should print its two interfaces
ip -n gwB -br addr show    # should print its two interfaces
ip -n hB -br addr show    # should print its one interface
```

1.5 Static routes across the WAN

Each gateway must be instructed on how to reach the opposite LAN.

```
# Assign routes
ip -n gwA route add 10.0.2.0/24 via 192.0.2.2 dev vAGwWan
ip -n gwB route add 10.0.1.0/24 via 192.0.2.1 dev vBGwWan
# Show routing tables
ip -n gwA route    # should print the added route
ip -n gwB route    # should print the added route
```

1.6 Add WAN delay, jitter, and loss

Simulates an 80-ms RTT Internet link with light loss to make the WAN connection similar to one in a real infrastructure.

```
# Add non-idealities
ip netns exec gwA tc qdisc add dev vAGwWan root netem delay 40ms 5ms loss 0.5%
ip netns exec gwB tc qdisc add dev vBGwWan root netem delay 40ms 5ms loss 0.5%
# Verify the addition of non-idealities
ip netns exec gwA tc -s qdisc show dev vAGwWan    # should print the new non-
    idealities
ip netns exec gwB tc -s qdisc show dev vBGwWan    # should print the new non-
    idealities
```

1.7 Plain-text connectivity test

Confirm routing and non-idealities settings.

```
# Simple ping operation to assess the connectivity
ip netns exec hA ping -c3 10.0.2.2
ip netns exec hB ping -c3 10.0.1.2
# Stress test to verify the non-idealities of the WAN connection
ip netns exec hB iperf3 -s -1 &
# On another shell with sudo privileges
ip netns exec hA iperf3 -c 10.0.2.2 -u -b 20M -t 10
```

2 StrongSwan (IKEv2/IPsec)

The StrongSwan configurations must be created for both *Gateway A* and *Gateway B*. The high-level idea is to define **one IKE_SA** (i.e., `s2s`) and one **CHILD_SA** (i.e., `net-net`) on each side. The subnets of the namespaces are mapped onto security properties that must be honored at connection time.

2.1 Certificates management

Build a minimal Certificate Authority for authentication purposes. It is necessary to have a private key *ca.key* that remains offline and issue a self-signed root certificate *ca.crt*. The gateways will trust it when they verify each other.

```
# Generate the CA's private key
pki --gen --type rsa --size 4096 \
    > ca.key
# Issue a self-signed root certificate (1 year validity)
pki --self --ca --lifetime 365 \
          --in  ca.key \
          --dn  "C=EU, O=Validation, CN=Validation CA" \
          > ca.crt
```

Give each gateway its own identity certificate with a key pair for each gateway. The CA signs a certificate valid for 30 days.

```
# Define the IDs for the gateways
for GW_ID in gwA gwB;
    do
        # Private key generation
        ipsec pki --gen --type rsa --size 3072 > "${GW_ID}.key"
        # Certificate signed by the CA
        pki --pub --in "${GW_ID}.key" \
          | pki --issue --lifetime 30 \
                --cacert ca.crt --cakey  ca.key \
                --dn  "C=EU, O=Validation, CN=${GW_ID}" \
                --san "${GW_ID}" \
                --flag serverAuth --flag ikeIntermediate \
                > "${GW_ID}.crt"
    done
```

Install certificates and keys where *swanctl* expects them, copying the CA root and moving each node's certificate and private key to the proper folders. Note that these folders are generated by default when installing strongSwan. Key permissions should be specified for security.

```
# Put the certificates for gateway A in the proper folders
cp ca.crt /etc/swanctl-gwA/x509ca/
mv gwA.crt /etc/swanctl-gwA/x509/
mv gwA.key /etc/swanctl-gwA/private/
# Put the certificates for gateway B in the proper folders
cp ca.crt /etc/swanctl-gwB/x509ca/
mv gwB.crt /etc/swanctl-gwB/x509/
mv gwB.key /etc/swanctl-gwB/private/
```

```
# Define read/write privileges
chmod 600 /etc/swanctl-gwA/private/gwA.key
chmod 600 /etc/swanctl-gwB/private/gwB.key
```

2.2 Secure channel configurations

Define the following configuration files at the indicated paths (create eventually missing folders). The separation of the files into two distinct paths mirrors how the configurations would live on the two separate hosts. Each gateway specifies its local and remote interfaces across the WAN connection. The child SAs refer to the two LANs, as demonstrated by the addresses. Note that these two files are a possible example, but the configuration can be customized according to what is intended to be tested.

Gateway A `/etc/swanctl-gwA/swanctl.conf`

```
connections {
    s2s {
        version = 2
        proposals = aes256gcm16-prfsha256-ecp256
        local_addrs = 192.0.2.1
        remote_addrs = 192.0.2.2
        fragmentation = yes
        dpd_delay = 40s
        mobike = yes
        encap = yes
        local {
            auth = pubkey
            certs = gwA.crt
            id = gwA
        }
        remote {
            auth = pubkey
            id = gwB
        }
        children {
            net-net {
                local_ts = 10.0.1.0/24
                remote_ts = 10.0.2.0/24
                esp_proposals = aes256gcm16-prfsha256-ecp256
                start_action = trap
                dpd_action = clear
            }
        }
    }
}
secrets {
    private_key_gwA {
        file = gwA.key
    }
}
```

Gateway B `/etc/swanctl-gwB/swanctl.conf`

```
connections {
    s2s {
        version = 2
        proposals = aes256gcm16-prfsha256-ecp256
        local_addrs = 192.0.2.2
        remote_addrs = 192.0.2.1
        fragmentation = yes
        dpd_delay = 40s
        mobike = yes
        encap = yes
        local {
            auth = pubkey
```

```
            certs = gwB.crt
            id = gwB
        }
        remote {
            auth = pubkey
            id = gwA
        }
        children {
            net-net {
                local_ts = 10.0.2.0/24
                remote_ts = 10.0.1.0/24
                esp_proposals = aes256gcm16-prfsha256-ecp256
                start_action = trap
                dpd_action = clear
            }
        }
    }
}
secrets {
    private_key_gwB {
        file = gwB.key
    }
}
```

2.3 StrongSwan configuration files

Two StrongSwan configuration files must be instantiated to instruct the *charon* daemon to look for a different base configuration compared to the default one (which would be at `/etc/strongswan.conf`). The default VICI socket would be at `/var/run/charon.vici`, but here it is replaced with a custom one for each namespace. This way, the daemons do not fight for the common one, guaranteeing isolation. Create the two following files at the indicated locations, copying the provided content.

Gateway A - `/etc/netns/gwA/strongswan.conf`

```
charon {
    plugins {
        vici {
            socket = unix:///run/charon-gwA.vici
        }
    }
}
```

Gateway B - `/etc/netns/gwB/strongswan.conf`

```
charon {
    plugins {
        vici {
            socket = unix:///run/charon-gwB.vici
        }
    }
}
```

2.4 Start charon & load configurations

Launch the daemon process inside the two namespaces and create a namespace-local UNIX socket for VICI control (using the file created in the previous step). This way, *swanctl* is able to talk to the correct daemon without concurrency fights. Then, load the configuration file into the running daemon using the newly created VICI socket and parsing the custom directory for the file.

```
# Launch the daemons for in both gateways
ip netns exec gwA charon-systemd --nofork &
ip netns exec gwB charon-systemd --nofork &
# Load the secure channel configurations
```

```
ip netns exec gwA env SWANCTL_DIR=/etc/swanctl-gwA \
    swanctl --load-all --uri unix:///run/charon-gwA.vici
ip netns exec gwB env SWANCTL_DIR=/etc/swanctl-gwB \
    swanctl --load-all --uri unix:///run/charon-gwB.vici
```

2.5 Bring tunnel up

Send a VICI request to **initiate** the daemon and the IKE_SA. The *net-net* CHILD_SA is set up within the *s2s* profile. This is done in the *gwA* namespace, so that *gwB* will respond, and the connection is established, installing two Security Associations and two Security Policies (one per direction). This operation could be performed by initiating the connection in the *gwB* namespace, and the outcomes would not change.

```
# Initiate from gwA
ip netns exec gwA env SWANCTL_DIR=/etc/swanctl-gwA \
    swanctl --initiate --child net-net --uri unix:///run/charon-gwA.vici
```

Verify that the configurations and the SAs have been correctly installed:

```
# Print the SAs, which should recall the configuration details
ip netns exec gwA swanctl --list-sas --pretty \
    --uri unix:///run/charon-gwA.vici
ip netns exec gwB swanctl --list-sas --pretty \
    --uri unix:///run/charon-gwB.vici
# Print the states, which should be mirrored according to the gateway
ip -n gwA xfrm state | head
ip -n gwB xfrm state | head
```

2.6 Functional test

The following connectivity test can prove that only *ESP* traffic is sniffed by *tcpdump*, demonstrating that the traffic has been secured.

```
# On a shell
ip netns exec hA ping -c3 10.0.2.2
# On another shell
ip netns exec gwA tcpdump -n -i vAGwWan udp -c 6   # should print 6 ESP captures
    encapsulated in UDP packets
```

2.7 Teardown

To **close the connection**, the *charon* daemons must be killed. Finally, the kernel XFRM tables must be flushed.

```
# Kill the charon daemons
pkill -x charon-systemd
# Flush residual states and policies
for ns in gwA gwB;
    do
        ip netns exec $ns ip xfrm state flush
        ip netns exec $ns ip xfrm policy flush
    done
```