**Network Security Project 2 Design Documentation**

Group 10

We have relocated the server functionality from the customers' local networks to the cloud using strongSwan, and changed the cloud network to use the private IPv4 address space 10.1.0.0/16.

**Test**

1. Start the vagrant VMs according to the project README.
2. As we use docker containers to start the server application in the background automatically, you can directly log into the clients to send messages.
3. Use "vagrant ssh client-xx" to log into the clients, then "cd client\_app && node client.js".
4. Use "vagrant ssh server-s1" to log into the server, then check the container id with "sudo docker ps", then "sudo docker logs xxxx(container id)" to check incoming connections.
5. Use "vagrant ssh router" to log into the router, then "sudo tcpdump -i enp0s8 -nvv", then check all the packets are encrypted by protocol ESP.

**Topology**

Diagram

Description automatically generated

**Routing**

1. We use NAT Masquerade for the gateway interface so the local ip addresses can not be leaked outside their own subnets.
2. We bind the original local server IP address 10.1.0.99 to the interface enp0s9 for gateway A and B so the ARP response for ip address 10.1.0.99 is available.
3. For gateway A and B, we redirect the traffic from the original local server address 10.1.0.99 of port 8080 to cloud gateway 172.30.30.30 of port 8080 with Destination NAT.
4. For gateway S, we redirect the traffic from the client gateway A and B (172.16.16.16 and 172.18.18.18) of port 8080 to server s1 address 10.1.0.2 of port 30000 and 30001 into the containers (different ports for scalability).

**Firewall**

1. We decided to use strict firewall rules on the clients, as there should be no need for the clients to visit the Internet. (We cache the client nodeJS dependencies into node\_modules.tar.gz and unpack when we first initialize the client)
2. We use iptables to setup firewall rules on gateway-a and gateway-b for both input and output.
3. First, we accept vagrant virtual machine traffic, then we accept IKE and ESP traffic (esp protocol, port 500 and 4500 with udp protocol) both from and to the cloud.
4. Finally, we drop everything else, this would include the connection from and to the Internet.

**PKI**

We generate the certificates based on the Elliptic Curve Digital Signature Algorithm (ECDSA). The certificate fingerprint is generated using the SHA512 hash function. The Root CA is valid for 10 years. The intermediate CA is valid for 5 years. The end-entity certificates are valid for 2.5 years. Except for the root CA certificate, each certificate is generated by first generating an ECDSA private key, then generating the Certificate signing request, and finally issuing the certificate from the private key and with CA. The details for certificates generation are as follows:

1. We generate the root CA. The subject of it is "C=FI, O=CSE4300, CN=CSE4300 Root CA". Put the root CA certificate into “/etc/ipsec.d/cacerts/caCert.pem”.
2. We generate the intermediate CA signed by the root CA. The intermediate CA is assigned with a pathlen parameter equal to 0. The subject of it is "C=FI, O=CSE4300, CN=CSE4300 INT CA". Put the intermediate CA certificate into “/etc/ipsec.d/cacerts/intCaCert.pem”.
3. Generate the end entity certificate signed by the intermediate CA. For site A, the subject is "C=FI, O=CSE4300, CN=CSE4300 Site A 172.16.16.16", site B is "C=FI, O=CSE4300, CN=CSE4300 Site B 172.18.18.18", cloud is "C=FI, O=CSE4300, CN=CSE4300 Cloud 172.30.30.30". We set the serverAuth flag for the cloud and the clientAuth flag for sites A and B. Store the site A certificate in "/etc/ipsec.d/certs/siteACert.pem", site B certificate in "/etc/ipsec.d/certs/siteBCert.pem", cloud certificate in "/etc/ipsec.d/certs/cloudCert.pem".
4. Generate the Certificate Revocation List (CRL) for both the root CA and the intermediate CA, and store them into "/etc/ipsec.d/crls/”
5. Finally, we store the secret keys of root CA and intermediate CA in the vault. Each gateway has a copy of the root CA certificate and intermediate CA certificate, the secret key that belongs to them, and the end entity certificates for both sides that will be used when communicating.

In conclusion, our PKI architecture has one root CA and another intermediate CAs, where the private key of the intermediate CA is used to sign the end entity certificates and the private key of the root CA can be kept on a smartcard stored in a safe or at lease on a system disconnected from the Internet. The private root CA key never stores on an insecure or online system. Securing the root CA enables the PKI administrator to revoke any certificates and recreate the PKI from scratch if any intermediate CAs are compromised.

**IPSec**

1. We set the IKEv2 as our key exchange protocol.
2. We select our ciphers according to the Commercial National Security Algorithm (CNSA) Suite where the strongSwam mentioned in their security recommendations: [https://docs.strongswan.org/docs/5.9/howtos/securityRecommendations.html#\_cipher\_selection](https://docs.strongswan.org/docs/5.9/howtos/securityRecommendations.html" \l "_cipher_selection), which is, we use aes256gcm16-prfsha384-ecp384 for the IKE and aes256gcm16-ecp384 for the ESP.
3. We use static IP addresses, store the certificates of both sides in the gateway in advance, and do identity and CA checks when authenticating in case of the man-in-the-middle attack.
4. We set the VPN connection to load a connection and bring it up immediately when started up.
5. We set dpdaction to hold, so empty INFORMATIONAL messages (IKEv2) are periodically sent to check the liveliness of the IPsec peer. A trap policy is installed, which will catch the traffic when the IPsec peer is dead and tries to re-negotiate the connection on demand.
6. We enable the Perfect Forward Secrecy (PFS) by following the default. It makes IPsec peers negotiate an independent session key for each IPsec or CHILD SA. This would protect the long-term confidentiality of the IPsec traffic if the IKE shared secret is leaked. The session keys of the first CHILD\_SA of a new IKEv2 connection are derived from the IKE shared secret. However, subsequent CHILD\_SAs will use independent keys if PFS is used.
7. We enable the MOBIKE protocol by following the default as well, MOBIKE allows that, if the configuration changes, route lookups are done to find a better path than the current one and, if necessary, the path is changed using a MOBIKE update (UPDATE\_SA\_ADDRESS).