Authors: Floran NARENJI-SHESHKALANI (643166) & Jean-Marcellin TRUONG (643357)

1 General idea

The original design is composed of several IoT clients exchanging with a single server on the same local area network (called IoT LAN further on). The stated objective is to move that server to the cloud while keeping the communications secure using a site-to-site VPN. Beyond safety, one of the main design goals is to make this architecture scalable in the sense that supporting more customers (more IoT LANs/server pairs) should be straightforward.

From a technical standpoint, the servers in the cloud have their own local area network (called cloud LAN further on) that is not connected to the internet.

2 Proof of Concept

This proof of concept will contain two customer networks: there will be two IoT clients, two IoT gateways, one cloud gateway and two cloud servers. The first customer is called 'A' and the second one 'B'.

2.1 Docker

While the scenario intends for this setup to be used in a corporate setting, the environment has for obvious reasons to be simulated. For that purpose, **Docker** is used. Docker offers a quick and easy way to setup and script multiple simulated machines. The main advantage of Docker over traditional virtual machines is ease of reproducibility on different setups, which is key when doing groupwork.

Moreover, Docker is commonly used for cloud-based setups meaning that, for the cloud part, our simulation is somewhat similar to a real-world application.

Docker Compose is used on top of Docker. Docker Compose is a tool for managing container instances: containers and networks are programmatically declared and the infrastructure as a whole can then be managed using simple commands.

All instances and configuration of the instance can be found in the `docker-compose.yml` file. The images ran by the containers are situated in subfolders each containing a `Dockerfile`.

2.2 IoT device & server

The specifications state the limitations of the IoT device but nothing about it's actual function. Therefore, the simplest solution was chosen: a simple bash script runs inside its own Docker container and simulates the IoT client. This script uses **curl** to send a plain-text HTTP GET request containing the current timestamp to a specified IP address/port.

The server is a simple stateless **python** script running inside its own container. The server receives the data from the client over HTTP, prints it out and discards it.

Note that, due to a Docker technical limitation (host machine is seen as the default gateway), the default route of both client and server is manually overwritten to the correct gateway (the local VPN server).

The IoT device image can be found in the `iot` subfolder. The server image can be found in the `server` subfolder.

2.3 Gateways

The IoT gateways (one per customer network) each have their own tunnel to the cloud gateway, which is common to all customers. The gateways serve as the main routing nodes of their LAN network, and all traffic goes through them.

They are also set up to only let through traffic that is relevant for each network (see Section 3.6 for the firewall rules).

The image for the client gateways is in the `gw` subfolder. For each instance of this image, there exists a subfolder (`agw/` and `bgw/`) which contains the certificates and key for the instance. Through Docker Compose, this folder is mounted in the instance.

The cloud gateway image is in the `cloudgw/` subfolder.

2.4 Usage

To run the infrastructure, `docker` and `docker-compose` must be installed. Inside the folder, the following commands can then be used:

```
# Build the images
$ docker-compose build

# Launch the gateways in the background
$ docker-compose up -d cloudgw agw bgw

# Launch the servers
$ docker-compose up asrv bsrv

# In another terminal, launch the client
$ docker-compose up aiot
$ docker-compose up biot
# Each client will make a request to its server and to the other server
# The one to the other server should fail.

# To shell into a gateway, e.g. the cloud one
# Note that exiting the shell will not stop the container
$ docker-compose exec cloudgw zsh

# Inside the container, you can use the following commands to observe the
# status of the tunnels and the firewall rules
$ > ipsec statusall
$ > iptables-save # To shut down the infrastructure
$ docker-compose kill

# To generate a new PKI (overwrites the previous, have to rebuild after)
$ ./genpki.sh
```

3 Technical choices

3.1 VPN software

Various options were considered for the VPN software, but it finally boiled down to picking between OpenVPN and Strongswan. Both pieces of software are well-suited to the usecases and are equally strong and safe.

First of all, we already have experience in setting up OpenVPN on our own dedicated servers.

Then, Strongswan implements **IPSec**, an industry standard that is quite widespread on common routing infrastructure. A client who would like to use its own router could readily configure it so that it connects to our VPN (such configuration is outside the scope of this project but can be done by a network administrator), which is a good selling point for the customer and a decrease in cost for the company.

Thus, **Strongswan** was chosen as it was both a sound choice and a good learning experience.

3.2 Authentication (initial design)

⚠ **This part talks only about our initial design from the first design document.** The final design is detailed in the next part. It is left here because the comparison between PSKs and PKIs was interesting research for us.

For authentication, a public key infrastructure with a root certificate and one certificate for each of our gateways was the initial choice. Unfortunately, we had not yet managed to get it working by the initial design deadline and authentication systematically failed.

Consequently, Pre-Shared Key (**PSK**) authentication was used, in which the password is stored in clear-text in Strongswan's configuration file. At the time, because of the scenario's simplicity (only two gateways were simulated), we believed this to be as easily maintained as a PKI-based authentication solution (more on that later).

In terms of security, we believe both solutions to be equivalent in most regards: if the machine is compromised, then a PSK or a private key can be equally extracted, and both can then be replaced easily by an admin after the breach. If the private key is password protected and the password is not provided in the configuration files, Strongswan will ask for it on startup. The same also applies to PSKs if so configured.

For an established connection, having access to the PSK is not a security concern as it is only used for initial authentication and not as a session key.

In the end, however, a PKI is vastly superior for multiple reasons. First of all, an existing organization is likely to have its own PKI already, and it would make sense to leverage such an existing system.

Then, while the current needs are quite simple and are easily satisfied by a PSK, future needs might not be: there might be more sites, more clients and potentially a need for revocation. Such an evolution in requirements is clearly a usecase in favor the PKI: it provides a tremendous advantage in scalability, maintenance and security.

Additionally, in a configuration with multiple customers, a successful attack against the cloud gateway entails a full replacement of all PSKs, whereas it would suffice to provide a new certificate and key pair when using a PKI. Even safer, a smartcard can be used to hold the private key, in which case it should be secure against both physical and remote attacks.

Finally, a key provided through a well configured PKI will always be cryptographically safe, whereas a PSK chosen by sloppy students might only be 7 characters long (when it should have been a very long string of maximum entropy).

3.3 Encryption algorithms

Strongswan accepts 3 paramaters related to communications security.

For encryption, AES256-CBC is used, as it is a well-known standard and provides included MAC.

For integrity, prfsha512, a sha512 hash combined with a pseudorandom function, is used.

Finally, the session key is constructed using ECDH with curve ed25519. This provides perfect forward secrecy, meaning that obtaining the private key used for authentication will not allow an attacker to compromise an ongoing connection.

3.4 Authentication (final)

For the final design, the **public-key infrastructure** is up and running. The keys are generated outside of any Docker container (e.g. this is the simulated equivalent of having an offline machine where the private key for the root certificate is stored).

The private keys are generated using curve ed25519 and the certificates are signed with ECDSA.

Each IOT gateway has its own key and certificate, the root certificate and the cloud gateway's certificate.

The cloud gateway has its own key and certificate, the root certificate and all certificates from the other gateways.

Note that checking only the name of the other party's certificate would have been sufficient, but checking both name and certificate offers additional safety (e.g. if the root certificate's private key is compromised but not the individual private keys, then the certificates should remain safe to use until replacement). This practice is generally known as **certificate pinning**, which is safer but requires more administrative work.

3.4.1 Safety concerns over private key storage

For the prototype, the gateway containers are preloaded with the certificates and key. In a real-world application, delivering keys through an API seems like an unnecessary safety risk: it offers a larger attack surface for a

part of the infrastructure that is highly critical. Designing and building such a system lays outside of the scope of this project, and is probably best left to specialists.

However, there remains a dual concern regarding safe private key delivery and storage.

As an answer to both those concerns, Strongswan supports smartcards. For a real-world application, it is our utmost recommandation that smartcards be used.

A **smartcard** is a very safe way to get a private key to the customer's physical site: smartcards are believed to be physically tamperproof and would not fall victim to a remote attack. The smartcard could then be delivered to the client and the gateway remotely configured, implying less time spent, reduced transportation fees and therefore better rentability.

3.5 Routing

For each customer network, a separate Docker container runs the back-end server. Such a setup allows efficient use of the resources of the cloud servers: a Docker container has a very small resource footprint and the application is presumed not to be computationally heavy.

It also avoids systematic use of a reverse proxy: because each container has its own IP in the cloud LAN, the IoT devices can contact its target directly.

From a security standpoint, we can ensure using the firewall on the cloud gateway that communications are only flowing between devices belonging to the same customer (cf Section 3.6).

Note that, should the load become too important for a single server, there are multiple server-side options (ignoring software optimizations, sorted by order of preference):

- the container could be moved to its own cloud server (instead of sharing with other containers)
- the container could be moved to a more powerful cloud server
- a load-balancer could be added in front of a cluster of containers running on different machines

3.6 Firewall

On the cloud gateway, the default iptables policy is to forbid all traffic. Packets used by the IPSec protocol itself are then allowed on the internet-facing interface only.

The key rules, of which there is a pair for each IoT network, are the following:

```
  -A FORWARD -p tcp -s $LAN_SOURCE/16 -d $DEST_SERVER/32 -dport 80 -i eth_internet -o eth_lan
-m conntrack -ctstate NEW,ESTABLISHED -m policy -dir in -pol ipsec -reqid $TUNNEL_ID -proto
esp -j ACCEPT

  -A FORWARD -p tcp -s $DEST_SERVER/32 -d $LAN_SOURCE/16 -sport 80 -m conntrack -ctstate
ESTABLISHED -i eth_lan -o eth_internet -m policy -dir out -pol ipsec -reqid $TUNNEL_ID -proto
esp -j ACCEPT
```

The first rule will allow incoming traffic from the IoT LAN and to the destination server (and only that one server) on the TCP port 80 (HTTP) coming in from the internet and going towards the cloud LAN if it has travelled through the correct IPSEC VPN tunnel (reqid parameter).

Let's go through each parameter of the rule:

-A FORWARD Packets being forwarded (not packets that are directly for/from the gateway)

-p tcp -s $LAN_SOURCE/16 -d $DEST_SERVER/32 -dport 80 Packets containing a TCP payload coming from the $LAN_SOURCE subnetwork and destinated to the $DEST_SERVER on port 80 only.

-i eth_internet -o eth_lan Packets coming in on the internet interface and going out on the cloud interface.

-m conntrack –ctstate NEW,ESTABLISHED Packets who are opening a new TCP connection or belonging to an existing TCP connection (stateful tracking of TCP connections).

-m policy –dir in –pol ipsec –reqid $TUNNEL_ID –proto esp Inbound IPSec packets coming from the IPSec tunnel identified by $TUNNEL_ID

The second rule is the same but in the other direction.

Furthermore, both rules have statefulness constraints: traffic can only be initiated from the IoT LAN to the cloud LAN and traffic can only leave the cloud LAN through an already established connection.

On the LAN gateway, the rules simply check that traffic has proper origin and destination network and that it is indeed IPSEC traffic. As safety-related rules are already enforced on the cloud gateway, those rules are much more lax. Generally, as the IoT gateways are in the client's hand, they should not be trusted which explains why the cloud gateway has bigger responsibilities.

Firewall rules in full can be found in `gw/fw` for the IoT gateways and `cloudgw/fw` for the cloud gateway.