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 mail design goals is to make this architecture scalable in the sense that it would be easy to support other customers (e.g. other networks of IOT devices who want to securely contact a cloud server).

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

2.1 Docker

While the scenario intends for this setup to be used in a corporate environment, we have for obvious reasons to simulate such an environment. For that purpose, we chose to use Docker containers, which allow us a quick and easy way to setup multiple simulated machines and to script them to our needs. The main advantage of Docker over traditional virtual machines is ease of reproducibility. Moreover, Docker is commonly used for cloud-based setups, meaning that, for the cloud part, our simulation is somewhat similar to what it would be in a real-world application.

2.2 IOT device & server

The specifications state the limitations of the IOT device but nothing about it's actual functions. Therefore, believing this to be of no interest, we have created a simple bash script that is running inside its own Docker container as a way to simulate the IOT client. This script uses curl to send a plain-text HTTP GET request to a specified IP address/port containing the current date.

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 does nothing more with it.

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

2.3 Gateways

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

The gateways are set up to only let through traffic that is relevant for each network (see Section 3.3 for the firewall rules).

3 Technical choices

3.1 VPN

We considered various options for the VPN software, but it finally boiled down to picking between OpenVPN and Strongswan. It seemed that both softwares could satisfy our usecases and seemed equally strong and safe.

First of all, we already had past experience 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 is quite standard), which is a good selling point for the customer and a decrease in cost for our company.

Thus, we elected to use Strongswan as we felt it was both a sound choice and a good learning experience.

3.1.1 Authentication (initial design)

⚠ This part talks only about our initial design from the first design document. The actual design is detailed in the next part.

For authentication, we initially wanted to set up our own public key infrastructure such that we would have a root certificate and one certificate for each of our gateways. Unfortunately, as of now, we have not yet managed to get it working and authentication systematically fails, even though we believe all the certificates are signed adequately.

Consequently, we've fallen back onto onto Pre-Shared Key (PSK) authentication, where the password is stored in clear-text in the Strongswan's configuration file. Because our VPN usage scenario is so simple (there will be at most two participants in the VPN), we believe this to be as easily maintained as a PKI-based authentication solution.

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 extracted in the same manner, and both can then be replaced easily by an admin. If the private key is password protected, Strongswan will ask for it on startup, but the same can also be applied to PSK. For an ongoing connection, having access to the original mean of authentication is useless as it is only used for initial authentication and not as a session key.

However, a PKI would still be best for multiple reasons. First of all, an existing organization is likely to have its own PKI already, and it would be best to leverage such an existing system instead of "rolling one's own" through a PSK.

Then, while our current needs are quite simple and are easily satisfied by a PSK, our future needs might not be: we might have more sites, more clients and potentially a need for revocation. Such a change in need is clearly a usecase of the PKI: it provides a tremendous advantage in scalability and maintenance.

Additionally, in a configuration with multiple customers, a successful attack against of the cloud gateway compels a full replacement of all PSKs, whereas it would suffice to replace the certificate and key when using a PKI.

Finally, a key provided through a PKI will always be fairly long, whereas a PSK chosen by sloppy students might only be 7 characters long (where it should be a very long random string).

3.1.2 Authentication (final)

For our final design, we have a set up a public-key infrastructure. The keys are generated outside of any Docker container (e.g. this is our equivalent of having an offline machine where the private key for the root certificate is stored). The Docker containers are preloaded with their key: there's no runtime mechanism for obtaining the key dynamically through an API or some such. Indeed, we feel that having such an API is a safety risk (bigger attack surface) and that it doesn't bring actual value to the application.

Currently, each gateway stores it's own private key on its drive, which feel is a potential danger in case of a breach. However, Strongswan supports smartcards and for a real application, we believe that distributing smartcards is one of the safest ways to get a private key to the customer's site: the smartcards are believed to be physically tamperproof (we could even consider posting them to the client and doing the whole setup remotely) and would not fall victim to a remote attack.

3.1.3 Encryption algorithms

Strongswan accepts 3 paramaters related to communications security. For encryption, we picked AES256-CBC for its MAC properties.

For integrity, we picked prfsha512, which is sha512 combined with a pseudorandom function.

Finally, we use elliptic curve based Diffie-Hellman with curve25515 for perfect forward secrecy.

3.2 Routing

For every customer network, we host a Docker container that runs the customer's back-end server. Such a setup allows us to efficiently use the resources of our cloud servers: a Docker container has a very small baseline resource cost and our application is not computationally heavy anyways.

It also avoids using a reverse proxy: because each container has its own IP in the cloud LAN, the IOT devices can contact it's target directly.

Moreover, this is also an advantage from a security standpoint: for each VPN tunnel, we can ensure using the firewall on the cloud gateway that communications are only flowing between the devices belonging to the same customer.

Note that, should the load become too important for a single server, there are multiple options (ignoring software optimizations, sorted by order of preference): the container could be moved to its own cloud server (instead of sharing with over containers), it could be moved to a more powerful cloud server or a load-balancer could be added in front of it in order to scale it out.

3.3 FIREWALL

On the cloud gateway, the default action is drop all packets Packets flowing on ports used by IPSec on the internet-facing interface are allowed.

The key rules are the following (there is one for each IOT network):

```
 -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).

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

Furthermore, both rules have statefulness constraints: traffic can only be initiated by 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 correct IPSEC traffic. The rules are much simpler because they are enforced already on the cloud gateway, for ease of maintenance and finally because the LAN gateway is in the hands of client, which means it cannot be trusted.