SIRS Project Report

MEIC-T, Tagus Park, Group 05

Afonso Bernardo (96834), César Reis (96849), Henrique Vinagre (96869)

I. Business Context

This project focuses on the company *EcoGes*, a new Electricity provider for customers' homes.

This new system grants near real-time (minute by minute) energy cost monitoring services, balancing the energy consumed by household appliances and that is produced by solar panels (if any), as well as allowing clients to manage the contract status and calculate energy invoices for the month, the current energy plan, and taxes.

II. Infrastructure Overview

The infrastructure for this application will consist of two servers, one serving the clients directly (let's call it the Application Server) and the other serving as a database (for the Application Server). We also have 3 departments, but we chose not to implement them, since they are not essential for our demonstration.

The Application Server will be located in a DMZ (192.168.1.0/24) connected between the external network (192.168.0.0/24) and internal network (192.168.2.0/24) by a gateway, which will act as a firewall between these two networks and the DMZ. On the other side, the Database Server will consist of an internal machine. Finally, the clients will be located in the external network, directly connecting to the firewall Gateway Server that enables port forwarding of the client's HTTP(S) requests into port 3000, where the application is running, and also allows Internet access to the hidden machines (at **enp0s10** interface).

All of the 3 departments (account managing, accountability and marketing) are located inside the internal network together with the Database Server.

II.1 Network topology

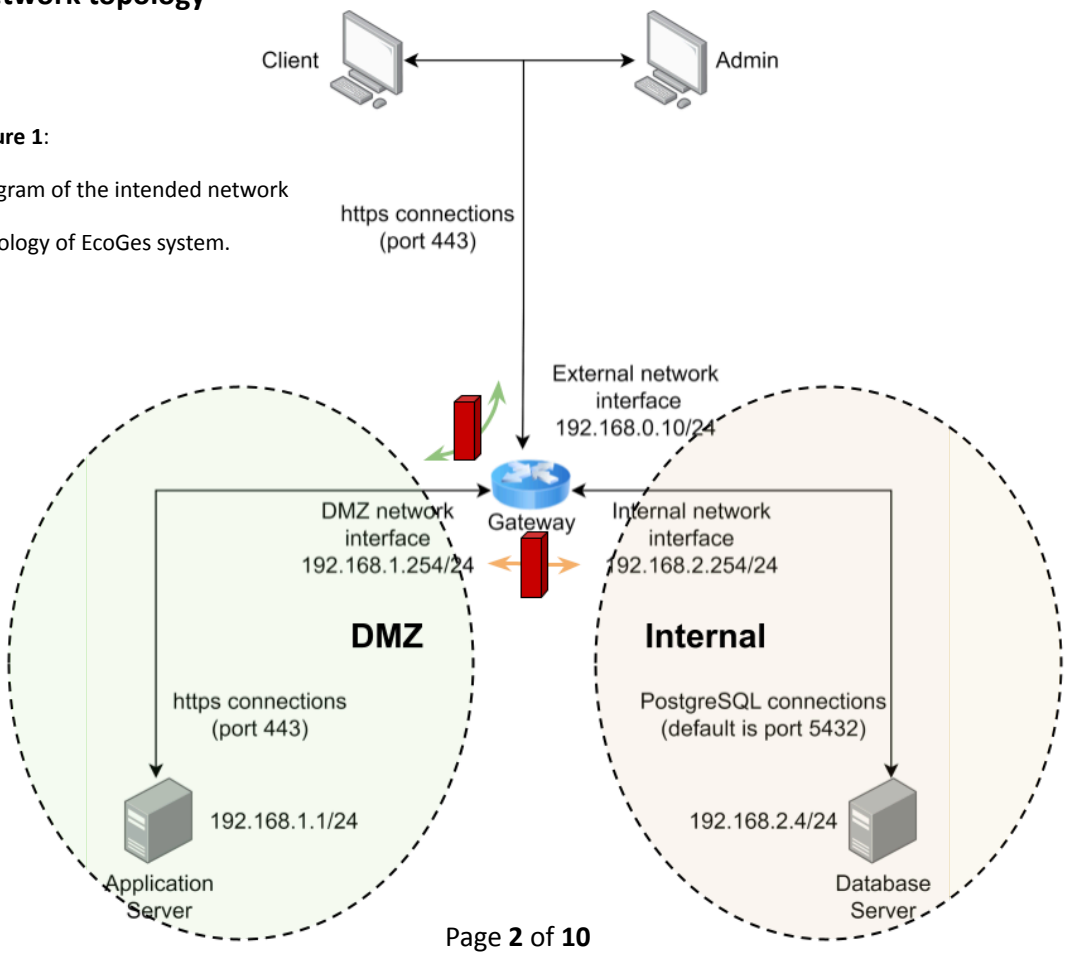

Figure 1:

Diagram of the intended network topology of EcoGes system.

II.2 Firewall rules

Suppose that **enp0s3**, **enp0s8** and **enp0s9** correspond to the external network, DMZ network and internal network interfaces of the gateway, respectively.

- sudo iptables -t nat -A PREROUTING -p tcp -i enp0s3 --dport 443 -j DNAT --to-destination 192.168.1.1:3000;
- sudo iptables -t nat -A POSTROUTING -o enp0s10 -j MASQUERADE;
- sudo iptables -P FORWARD ACCEPT;
- sudo iptables -A FORWARD -m state --state ESTABLISHED -i enp0s10 -j ACCEPT;
- sudo iptables -A FORWARD -m state --state NEW -o enp0s10 -j ACCEPT;
- sudo iptables -A FORWARD -m state --state ESTABLISHED -o enp0s10 -j ACCEPT;
- sudo iptables -A FORWARD -p tcp -i enp0s3 -o enp0s8 --dport 3000 -m state --state NEW -j ACCEPT;
- sudo iptables -A FORWARD -p tcp -i enp0s3 -o enp0s8 --dport 3000 -m state --state ESTABLISHED -j ACCEPT;
- sudo iptables -A FORWARD -i enp0s8 -o enp0s3 -m state --state ESTABLISHED -j ACCEPT;
- sudo iptables -A FORWARD -p tcp -i enp0s8 -o enp0s9 --dport 5432 -m state --state NEW -j ACCEPT;
- sudo iptables -A FORWARD -p tcp -i enp0s8 -o enp0s9 --dport 5432 -m state --state ESTABLISHED -j ACCEPT;
- sudo iptables -A FORWARD -i enp0s9 -o enp0s8 -m state --state ESTABLISHED -j ACCEPT;
- sudo iptables -A FORWARD -j DROP;
- sudo iptables -A INPUT -p tcp --dport 22 -j ACCEPT;
- sudo iptables -A INPUT -p tcp --dport 443 -j ACCEPT;
- sudo iptables -A INPUT -j DROP

II.3 Technologies

The Application Server will promote a Flask API (written in Python), providing the *EcoGes* business services which query the Database Server that will primarily integrate the PostgreSQL DBMS, which is responsible for managing the app user authentication and access to the database that stores the application's domain data.

The Web-application clients must access the website from a browser executing HTTPS requests to the Gateway Server. Uniquely, the system administrators can also operate remotely on the back-office machines through SSH. Lastly, the internal communication between the Application Server and the Database Server will be secured by a secure channel using TLS protocol to be integrated into the PostgresSQL remoting system, running on default port 5432.

III. Secure Communications

Channel1: The client (whether it is a regular user or an employee) will be communicating with the Application Server using TLS over HTTP (HTTPS).

- Client: The browser manages the client's public and private keys distribution and certification validation by itself.
- Applicational Server: Holds a certificate (**server.crt**) that is proposed for the client connection. This certificate is signed by a trusted CA, which is "registered" in the root CA (its certificate is **ca.crt** and private key **ca.key**). In our implementation, the CA is just a fictitious CA, with a self-signed certificate (this is because there is no use in paying for a certificate just to use in a demonstration). The app server must also contain a public key (**public.key**) and not disclose its respective private key (**server.key**). This way the client browser should trust the server certificate assuming that it recognizes the CA certificate.

Channel2: The Application Server must also establish a connection to the Database Server using an SSL/TLS certificate authentication.

- Applicational Server: Keeps both its public and private keys.
- Database Server: Contains a self signed certificate (**db.crt,** again it should be signed by a trusted CA, but for demonstration purposes we self-sign it), its private key (**db.key**) and a public key (**db_pb.key**). These key files are seen by the PostgresSQL configuration files (pg_hba.conf, for example) to allow the secure communication with the Applicational Server on the *EcoGes* database.

Channel3: The departments all have a secure connection with the Application Server. These are not implemented, so neither are the channels.

IV. Security Challenge

Security challenge: **i)**

Description of the challenge: "*The users can update their information but cannot see the critical data, such as **bank account ID** and **authorization**, **address**, and other personal information. The information is stored in the same physical database, but the data separation must be enforced by the use of cryptography."*

What is the main problem being solved?

Users shouldn't be able to get critical data, such as bank account ID, because it would be a security risk. In the same way, each department within the company should only see the part of this information they need for their activity.

Which security requirements were identified for the solution?

R1: Users should only be able to see their own information.

R2: The information accessed by users should be non-critical (only critical information is encrypted).

R3: Each department should only have access to the user-critical data that is crucial to its operation (the non-critical data is "public", as in anyone in the internal network can ask for it).

V. Proposed Solution

Who will be fully trusted, partially trusted, or untrusted?

Fully trusted:

- Admin

Partially trusted:

- Users with registered account and that are authenticated
- Employees

Untrusted:

- Anyone without a registered account or that don't pass the authentication phase

Trust relationship justification:

The admin should be able to access everyone's data, therefore being **fully trusted**.

The user should only be able to access his own non-critical data, thus being **partially trusted**, in the sense that the service only trusts them with a restricted scope of data.

The employees should only be able to access the data designated to their function, thus being **partially trusted**, in the sense that the service only trusts them with a partial scope of data.

Any other person should not be trusted with anything at all, therefore being **untrusted**.

How powerful is the attacker? What can he do and not do?

The attacker can try to pretend being another user (for example by spoofing the ip/cookie/etc…).

The attacker can intercept and retrieve the data being sent through the channel. Therefore, he can try a man-in-the-middle attack or a replay attack, etc…

The attacker can be an ordinary user with a registered account (or employee), so he can do everything a user can (update his own information and retrieve his own non-critical information).

The attacker can see the data being sent in the internal network, but can't modify (or block) it.

The attacker can't have access to any machine inside the internal network (i.e. he doesn't have control over the app server nor the database server).

Attacker model

Outside the company network (external network), the attacker can intersect the client session and be a man-in-the-middle, interrupting the client login and therefore, receiving the client requests and, more concerning, the EcoGes replies with confidential information or reply requests.

In the same way, if the attacker gets into the internal network, he can capture the database data flows of all active users by simply plugging in an Ethernet cable without needing to login with a user account.

As a regular user, the attacker, to receive admin privileges, may try to perform SQL injections on insert and update operations that the system promotes.

As an employee of a certain department, the attacker can try to get a hold of data that is not "assigned" to his department.

Secure protocol to develop

Before we describe the secure protocol view for our security challenge development, we clarify some assumptions that will be taken from now on:

Assumptions:

- The Application Server cannot be compromised. In other words, any file present in its file system that does **not** travel over the network cannot be captured;
- The company's back office is a distributed system, where the Application Server is centralised for any known department with all communications via secure channels;
- Each department has its own RSA private and public key. These don't necessarily need to be stored by each one of them, but we assume only the department (aka its employees) knows his private key.
- The departments are able to retrieve the intended data from the database as they are present on the same network as the Database Server (internal network);
- No employees are working in more than one department at the same time.

Protocol:

The encryption of the critical data will be performed by the Application Server when in operation. Initially, this server will randomly generate a secret key for each department in order to encrypt the user's critical data, separated by department. These secret keys are symmetric keys that in our implementation use the AES256 algorithm (with a random IV, i.e. initialization vector) on CBC mode with PKCS7 padding for encryption, but can easily be changed with other algorithms.

After this, the Application Server will establish a KEK for every department (the KEK can later be used to share a new key), this being the public key provided by the corresponding department. This KEK will be used to encrypt the corresponding department's secret key and later store the result in the Database Server through **Channel2**. In our implementation we chose RSA with PKCS1v15 padding for the encryption/decryption, but it can be changed.

When the Application Server receives the public key from a department (it can be sent in plain text through the secure channel) it must verify that it is indeed that department's key. This is done by checking with the trusted CA certificate (in this case it's the company's CA).

Each department will therefore manage its own pair of asymmetric keys signed by the CA of the company, being able to decipher its "assigned" secret key and nothing else (it can still see the public information).

For administration purposes, the Application Server checks if a user has admin privileges and, if it does, this server is able to retrieve all department's secret keys to give access to all the database information needed to retrieve without restrictions.

Who will communicate?

Our protocol regards the following communications:

- Application Server <-> Database Server
- Application Server <-> Departments
- Departments <-> Database Server

Which security properties will be protected?

The protocol that we will be implementing will ensure confidentiality for all known departments, and the secure channels previously established will provide authenticity (integrity + freshness).

Perfect forward secrecy doesn't apply to our problem, since there isn't a notion of "session", the data is simply stored encrypted in the database. Even if an attacker finds out a department's private key, he won't be able to retrieve the secret key of that department, since he can't query the database (it's in the internal network) nor can he get it by listening to the channels, since these are encrypted using TLS.

Existent keys and its management and distribution

Our protocol essentially require the existence of the following keys:

- **Secret keys** for encryption/decryption of the critical data separated by department. The Application Server generates these keys and then distributes them for the corresponding department. This distribution is simply encrypting with the corresponding **department's public key** and storing the result in the database.
- **Key Encrypting Keys** which correspond to the **departments' public keys**. Each department is responsible for providing its **public key** to the Application Server. These keys must be signed by the company's CA.
- **Departments' private keys** which are used to decipher the **secret key**, which is encrypted in the database.

Ku - Department's Private Key **CAr** - Company CA Certificate Signing **Ks** - Department's Secret Key

Secure protocol collaboration diagram

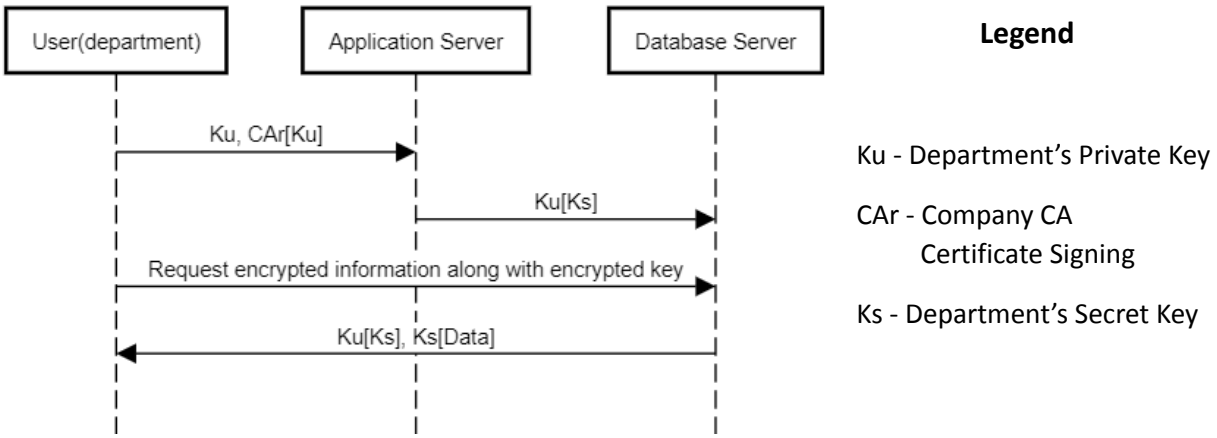

Figure 2 - Sequence diagram of the participating entities and the exchanged messages.

Libraries to use:

- cryptography (python 3.6+) for encrypting/decrypting in python;
- openssl for generating all asymmetric keys and certificates;
- psycopg2 for storing the keys in the database and for performing updates with essential encrypted data.

VI. Results

With our implementation we are able to conclude that:

R1: *satisfied* - The Application Server manages the user session separately, ensuring that the information not belonging to him is not seen in that session.

R2: *satisfied* - Users only have permission rights to (partially) see their own information. That is, only the non-critical information, as the remaining data is reserved to be seen only by the corresponding department).

R3: *satisfied* - Data is encrypted in function of the department it belongs to, thus only being accessible by the respective department. Any user public data is accessible within the internal network at all times without intervening any of these secret keys.

Main implementation choices

We opt to keep all the secret keys belonging to the departments at Application Server instead of removing them after storing in the database, since this server would ask departments for them every time a new update arrives.

We chose to not generate secret keys as ephemeral keys by the Application Server, since our security challenge doesn't imply care with perfect forward secrecy.

We use a pure cryptographic cipher with the cryptography python module for the department's secret keys to allow us to manage in a better way (at the white-box level) how we want to hide the user-critical data in the database properly.

As a department view, the Application Server behaviour is to reproduce the result of its activity to the user depending if it's a regular user where the critical data is omitted (not having access) or an admin seeing clearly all the results of its and all client operations.

VII. Conclusion

Our protocol allows for the database to be highly confidential, in the case that, for example, an attacker obtains full control of the database, no data should be deciphered, although it could be tampered (replaced by “garbage”, since the attacker cannot properly cypher the data they want to inject). For this same purpose, we use secure channels for communication that provide authentication, so the attacker shouldn’t be able to disrupt the database integrity either.

Although we implemented this solution towards the data separation between departments, this separation could be done between any other entity, following arbitrary criteria. This protocol is highly flexible for any data separation issues.

Possible future enhancements

Our key management is done statically, assuming that all departments’ servers are turned on at the moment that the protocol starts. There is room for improvement in this matter, as the internal network could have a key management server, which the departments could issue key generation requests once they are online, and could regenerate keys from time to time to ensure perfect forward secrecy.

References

- Flask api documentation
- Psycopg documentation
- PostgreSQL db documentation
- Python cryptography