Integrated MSc Course on Informatics Engineering, DI/FCT/UNL
Computer Networks and Systems Security / Semester 1, 2019-2020

WORK-ASSIGNMENT #2 REPORT for Evaluation

REPORT

Messaging Repository System

Summary

This handout was developed for the 2019 edition of the Computer Systems and Networks Security course in the FCT-UNL. The present work is an implementation of a secure messaging repository system using symmetric and asymmetric cryptography for data encryption over secure SSL channels.

The system was made to be almost fully parameterized for ease of testing and user freedom of choice. The system uses Diffie-Hellman public key cryptography to ensure the secrecy of all data exchanged between users. All the messages are fully encrypted, protected, and authenticated. The user and message receipt data are signed to ensure non-repudiation.

The system uses a Public Key Infrastructure to distribute, validate, as well as revoke certificates. This PKI is the root of trust of all the principals involved in the system. Certificate chains are used between the involved principals, to communicate over TLS secured channels, with the PKI certificate being the root certificate. Both the PKI and mailbox server resort to an SQLite local database to store all their data.

The mailbox server and clients have a cache implementation to prevent unnecessary, redundant communications. Many other systems are in place to raise overall system security by preventing Denial of Service, spoofing, replaying, sniffing, SQL injections and other types of attacks. A message exchange protocol was also created to allow for easy processing of message attachments (files), along with helper classes for all kinds of cryptographic operations.

Authors:

Filipe de Luna nº48425 e Miguel Candeias nº50647

Teacher:

Henrique João Domingos

Index

1. **Introduction………………………………………………………………………………2**

2. **System model and architecture…………………………………………………………3**

   2.1 Architecture………………………………………………………………………….3
   2.2 Threat model………………………………………………………………………..4

3. **Implementation details…………………………………………………………………….5**

   3.1 Keystores and Trust Store files……………………………………………………..10

4. **Work Evaluation and Validation…………………………………………………………11**

5. **Conclusion…………………………………………………………………………………..14**

Teacher: Henrique João Domingos
SRSC 2019/20 1ºSemestre
Filipe de Luna nº48425
Miguel Candeias nº50647

1. Introduction

In this second work of CSNS, we were requested to develop a secure message exchange system.

The system allows users to create message boxes, encrypt and send messages, receive and decrypt the messages, send and verify message receipts and list users, list all messages(sent and received) and the new messages and finally get a message status in order to check if the message was received or not, this means that no one else read it or and finally sending a receipt request that works as a proof that the message was well receipt and read.

Another particularity of this message repository is that it enables users to send more than just simple messages, they can also send data, meaning that users can also share files between them such as .txt, .jpeg, .mp4.

In order to do any other operation in the server, users must “log in”; in this case, logging in is more of a local authentication method that retrieves all keys from a user. Therefore, he must know his password to access key stores and trust stores and DH values that will be explained later in this report.

As mentioned in the summary, the message exchange system possesses a local database that stores all the user and message-related data.

Before communicating with the message server, users have to obtain a valid, signed certificate from the PKI. The PKI also acts as a Certificate Authority that the message exchange system can use to validate or revoke certificates.

2. System model and architecture

2.1 Architecture

Our system is split into two entities:

The Public Key Infrastructure (PKI) server that is responsible for issuing the user's certificate validating it and revoking it. Every entity involved in this architecture trust the PKI, meaning that all parties trust in the certificates issued and signed by this server.

This server acts as a trusting computing base (TCB) for all principals in the architecture.

It also validates, on request, certificates from users that requested a connection to access the mailbox. If the PKI does not validate the certificate from the user, the mailbox server denies him any access. The reason for the non-validation of the user's certificate can be due to its revoked status or due to it being expired.

The other entity is the mailbox server, where users can request operations such as creating a new account, sending and receiving messages, listing all messages, listing all users in the system, and checking the status of received messages' receipts from the other user.

Below is an example of the architecture and interactions:

2.2 Threat model

One of the adversaries in our threat model is a MiM (Man in The Middle) attacker that uses package snooping/sniffing tactics. To counter him, we resorted to using TLS communication channels to ensure authentication and encryption of all data exchanged in the OSI Application Layer.

Another adversary is the "honest but curious" administrator, which we protected by keeping all the message data in the server encrypted with keys that are not stored in the server and cannot be obtained by the server.

Another adversary is the SQL injection attacker from which we protected by using a Database Driver that allows transactions between the database and the server, and this protection comes from the use of Prepared Statements when making transactions in the database.

Another adversary is a system administrator with ill intent that wishes to corrupt user data or message data. Although he can make the message exchange stop working for users, he cannot change parameters, message contents, or receipts because they are all signed and authenticated with either a user's public key (for user data) or a shared Diffie-Hellman (DH) key (messages and receipts).

Another adversary, also a MiM adversary is one that wishes to pose as the server when responding to a user. We are protected against him by sending unique nonces in each client-server interaction that we expect to receive back from the server.
Another adversary is a hacker that can have access to the client's data on his computer. Although very hard to protect against such cases, we have some mitigation by using keystores and encrypting the files that contain the user's DH values with an AES cipher in GCM to ensure the data's confidentiality and integrity.
Another adversary model is the one that makes uses of replaying attacks, in this case, the system is protected from it by using the Secure transmission layer provided by TLS since TLS channels itself are protected from this type of attack.

Another adversary is the attacker that sends enormous files in order to clog and the server and provoke a Denial-of-Service. To prevent this, we have added custom Stream processing that is configured not to let uploads of more than x Megabytes be accepted.

3 Implementation details

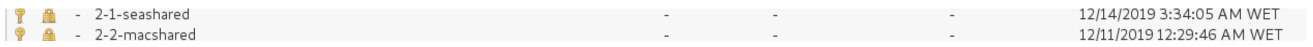
For our system, we developed two servers. The first one is a public key infrastructure or called the PKI server and our Certificate authority, as mentioned above. The PKI is the trusted computing base of all entities in this system. This server has three reasons for its use, and the first one is to sign CSR requests by users that wish to communicate with the mailbox server, creating a certificate chain with the PKI and client. After users receive the certificate from the PKI, they then use that certificate to authenticate TLS communications with the mailbox server. When users start communicating with the mailbox server, this server will communicate with the PKI in order to validate the certificate from the user, in case the certificate is revoked, the server will deny any access to it and users will have to get a new one. If the certificate is valid, they may proceed to access the mailbox's services. Another functionality of the PKI is the ability to revoke the certificates issued, therefore creating a revocation list.

The mailbox server uses a cache that temporarily saves certificates validated by the PKI, resulting in a faster service overall when the same client makes subsequent requests. The client implementation we have created also makes use of a cache that saves messages and users, for the same performance purpose. Messages maintained in the cache remain in their encrypted form for added security.

The first interaction the user has with the mailbox server is to obtain the shared parameters. These include all the DH parameters like key algorithm, key size, key hash algorithm, and public values as well as the public key algorithm used by the server to sign these parameters and by the users to sign their security information. These parameters can be reset at will by the server administrator by enabling the reset in the server's ".properties" file.

When creating a user, two Diffie Hellman Key Pairs are created by the user in order to register them in the server as public DH values. The first key pair is relative to the generation of a symmetric key between two users that want to send messages and want message confidentiality. The second key pair is used to generate a shared MAC key between the same users to guarantee message integrity. After generating the keys with another user's public DH values, the keys are encrypted and stored in a folder chosen by the user. The file name represents the users that the key references, as well as the respective mac/sea algorithm the key will be used for. The algorithms used to encrypt the files (AES in GCM mode) are hard-coded to avoid a user choosing a weak cipher since this file creation system was created to circumvent a limitation in the key store (not being able to save a private and public key), it is pretty much like a key store extension. It uses the same password as the key store to encrypt files. The DH public keys are then sent and stored in the server alongside the algorithms used for message encryption and MAC's used in client message traffic.

When sending a message to another user, the other user's DH public key is requested, in order to start the shared key agreement. After generating the shared keys, users create a shared symmetric encryption algorithm key and a shared mac key. This end to end process allows to have users privacy without any leak of information and risk of "honest but curious" individuals because no key is stored in the servers, these shared generated keys are then stored in the user KeyStore in a format such as the following example:



```
-  2-1-seashared      -   -   -   12/14/2019 3:34:05 AM WET
-  2-2-macshared      -   -   -   12/11/2019 12:29:46 AM WET
```

The clients automatically trim the shared DH keys as needed to fit the size needed for a SEA or MAC algorithm, and this is particularly necessary due to the fact that every user can specify which ciphers he wishes to use to encrypt the messages his sends.

The process of receiving a message is symmetric to sending one, and the user is requested to get the sender MAC and SEA specs and then generate the shared keys through the DH key agreement, after this the user can decrypt and thoroughly read the message. After reading the message, the user automatically sends a receipt message to the server, with the record date when it was read, all signed with the mac key shared between them.

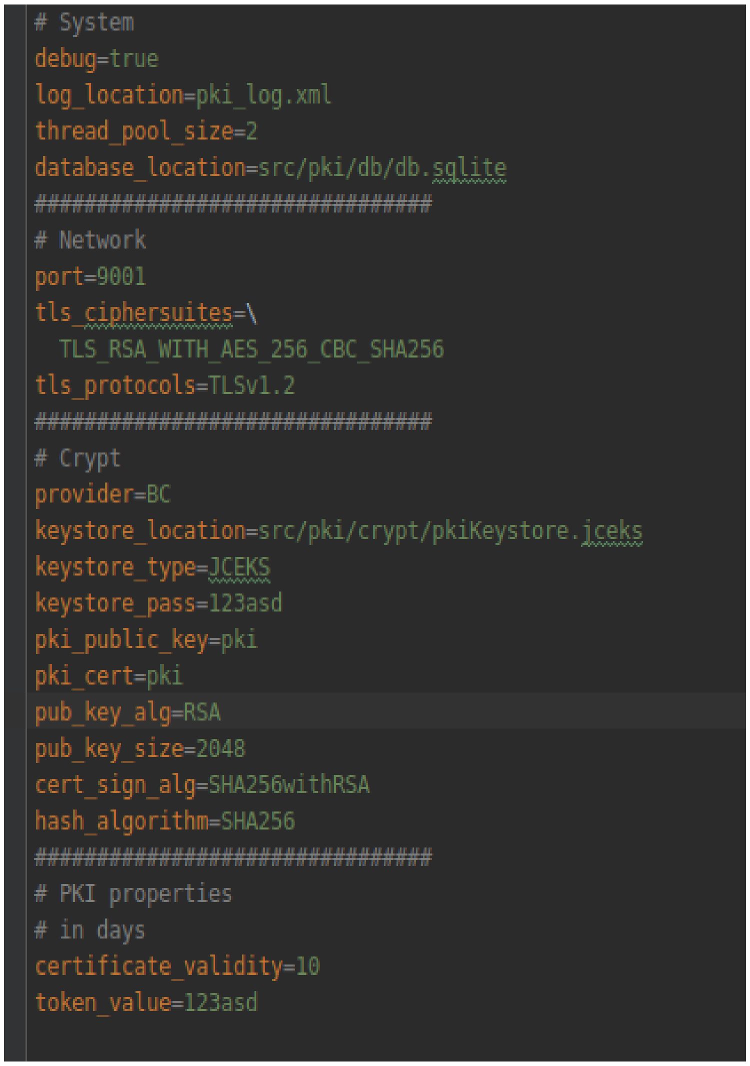
By forcing the client to authenticate using a valid certificate emitted by the PKI, we can associate certificates to actual people. This means that we can log suspicious behavior and easily find a culprit. Due to this fact, our PKI and Mailbox server heavily rely on logs to allow easy auditing of potential attacks. This makes even more sense since we envisioned a business model where a person would have to pay to get a token that can be exchanged for a signed certificate from the PKI, thus enabling him to use the service.

Besides just simple text messages, we designed a simple protocol that allows users to add attachments to the messages. These attachments can be any kind of file (txt files, pdf files, image files, and others). Since this could be exploitable, we added parameterized custom input streams that limit the max size of received data in the client and server, per interaction. Changing the max size of files sent is another parameterized value stored in the client properties file. Users can send more than one file at a time.

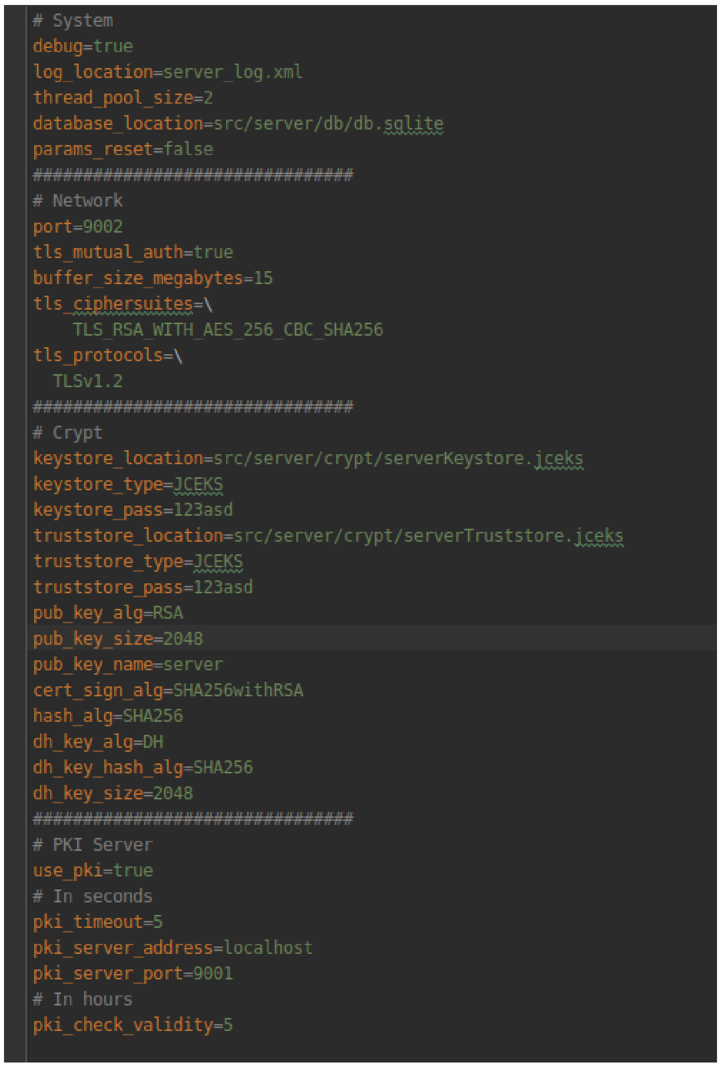
Another important aspect of our implementation is that almost everything is parameterized alongside with the extra things we implemented, such as the creation of a Thread Pool size in the server properties file and the PKI file. The use of a Thread Pool allows having both servers parallelized for dealing with a lot of concurrent requests in an efficient way. The validity of signed certificates by the PKI server is also another parameterized value in the PKI properties file.

TLS protocols and cipher suites, key stores, and trust store locations are also in all files allowing for much more flexibility between versions and TLS cipher specs.

PKI server and Mailbox Server parameter files have the following scheme respectively:

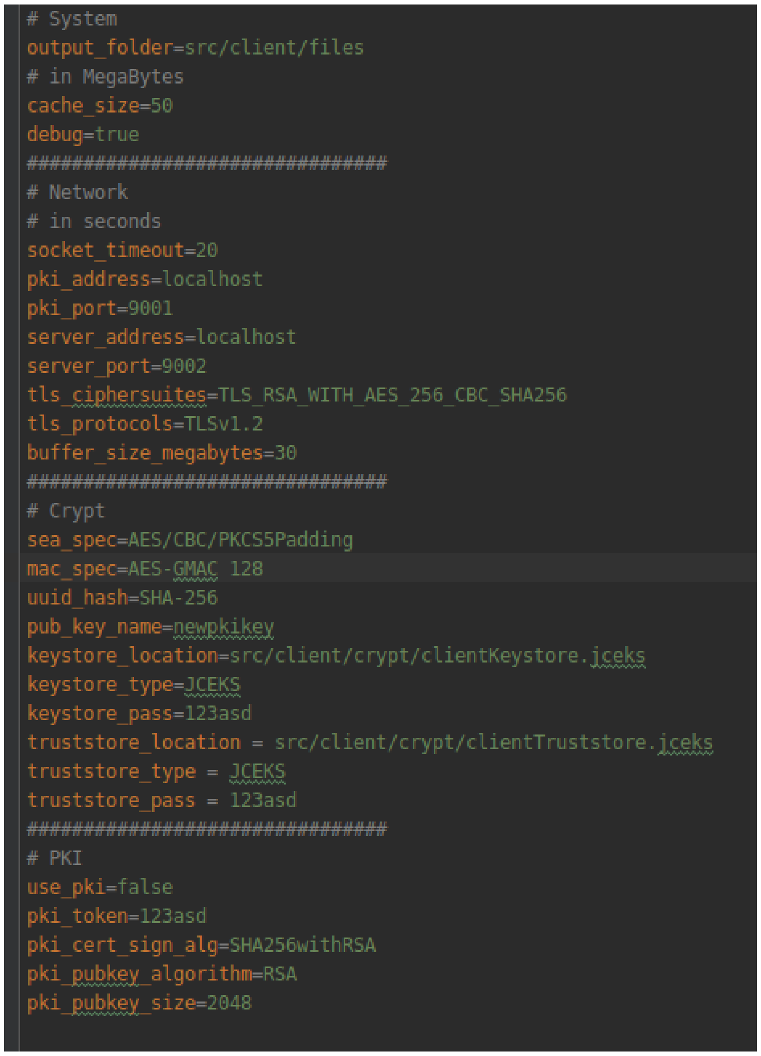


```
# System
debug=true
log_location=pki_log.xml
thread_pool_size=2
database_location=src/pki/db/db.sqlite
################################
# Network
port=9001
tls_ciphersuites=\
  TLS_RSA_WITH_AES_256_CBC_SHA256
tls_protocols=TLSv1.2
################################
# Crypt
provider=BC
keystore_location=src/pki/crypt/pkiKeystore.jceks
keystore_type=JCEKS
keystore_pass=123asd
pki_public_key=pki
pki_cert=pki
pub_key_alg=RSA
pub_key_size=2048
cert_sign_alg=SHA256withRSA
hash_algorithm=SHA256
################################
# PKI properties
# in days
certificate_validity=10
token_value=123asd
```



```
# System
debug=true
log_location=server_log.xml
thread_pool_size=2
database_location=src/server/db/db.sqlite
params_reset=false
################################
# Network
port=9002
tls_mutual_auth=true
buffer_size_megabytes=15
tls_ciphersuites=\
    TLS_RSA_WITH_AES_256_CBC_SHA256
tls_protocols=\
  TLSv1.2
################################
# Crypt
keystore_location=src/server/crypt/serverKeystore.jceks
keystore_type=JCEKS
keystore_pass=123asd
truststore_location=src/server/crypt/serverTruststore.jceks
truststore_type=JCEKS
truststore_pass=123asd
pub_key_alg=RSA
pub_key_size=2048
pub_key_name=server
cert_sign_alg=SHA256withRSA
hash_alg=SHA256
dh_key_alg=DH
dh_key_hash_alg=SHA256
dh_key_size=2048
################################
# PKI Server
use_pki=true
# In seconds
pki_timeout=5
pki_server_address=localhost
pki_server_port=9001
# In hours
pki_check_validity=5
```

The client properties file has the following schema:



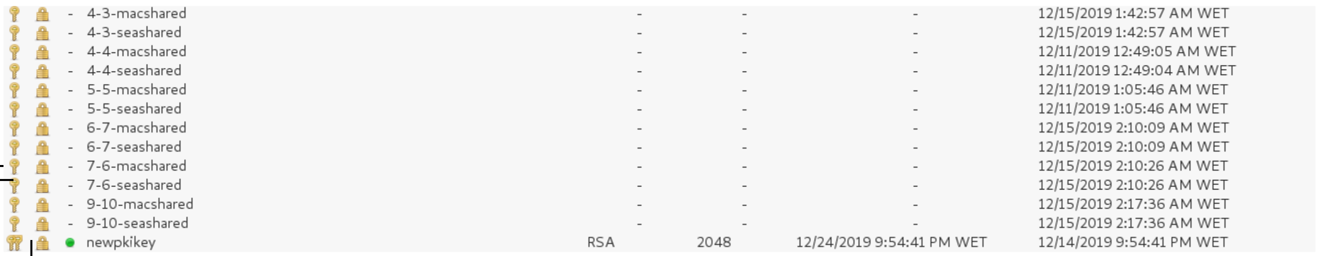
```
# System
output_folder=src/client/files
# in MegaBytes
cache_size=50
debug=true
################################
# Network
# in seconds
socket_timeout=20
pki_address=localhost
pki_port=9001
server_address=localhost
server_port=9002
tls_ciphersuites=TLS_RSA_WITH_AES_256_CBC_SHA256
tls_protocols=TLSv1.2
buffer_size_megabytes=30
################################
# Crypt
sea_spec=AES/CBC/PKCS5Padding
mac_spec=AES-GMAC 128
uuid_hash=SHA-256
pub_key_name=newpkikey
keystore_location=src/client/crypt/clientKeystore.jceks
keystore_type=JCEKS
keystore_pass=123asd
truststore_location = src/client/crypt/clientTruststore.jceks
truststore_type = JCEKS
truststore_pass = 123asd
################################
# PKI
use_pki=false
pki_token=123asd
pki_cert_sign_alg=SHA256withRSA
pki_pubkey_algorithm=RSA
pki_pubkey_size=2048
```

As we mentioned before, we also make use of prepared statements in our database driver classes, and this allows the security of the database against SQL Injection attacks.

We also developed a safe randomizer class to be able to reseed automatically every 50 times it is used. We can also change the mode between "Strong" and "Weak" allowing them to have extra security in the system.

3.1 Keystores and Trust Store files

User:

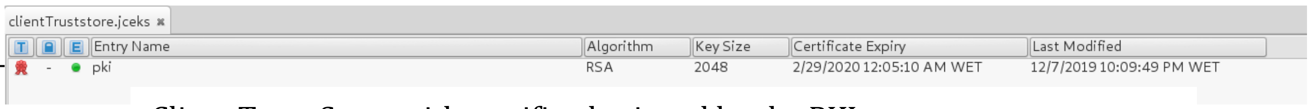


-	4-3-macshared	-	-	-	12/15/2019 1:42:57 AM WET
-	4-3-seashared	-	-	-	12/15/2019 1:42:57 AM WET
-	4-4-macshared	-	-	-	12/11/2019 12:49:05 AM WET
-	4-4-seashared	-	-	-	12/11/2019 12:49:04 AM WET
-	5-5-macshared	-	-	-	12/11/2019 1:05:46 AM WET
-	5-5-seashared	-	-	-	12/11/2019 1:05:46 AM WET
-	6-7-macshared	-	-	-	12/15/2019 2:10:09 AM WET
-	6-7-seashared	-	-	-	12/15/2019 2:10:09 AM WET
-	7-6-macshared	-	-	-	12/15/2019 2:10:26 AM WET
-	7-6-seashared	-	-	-	12/15/2019 2:10:26 AM WET
-	9-10-macshared	-	-	-	12/15/2019 2:17:36 AM WET
-	9-10-seashared	-	-	-	12/15/2019 2:17:36 AM WET
●	newpkikey	RSA	2048	12/24/2019 9:54:41 PM WET	12/14/2019 9:54:41 PM WET

Client key store with keypair generated from signed cert from the PKI

SEA spec generated shared key for users 7 and 6

MAC spec generated shared key for users 7 and 6

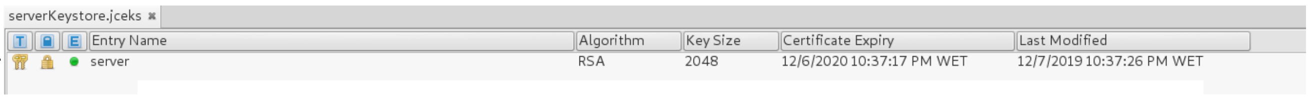


clientTruststore.jceks

T	E	Entry Name	Algorithm	Key Size	Certificate Expiry	Last Modified
-	●	pki	RSA	2048	2/29/2020 12:05:10 AM WET	12/7/2019 10:09:49 PM WET

Client Trust Store with certificade signed by the PKI

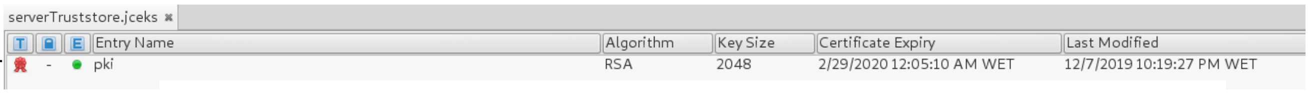
MailBox Server:



serverKeystore.jceks

T	E	Entry Name	Algorithm	Key Size	Certificate Expiry	Last Modified
	●	server	RSA	2048	12/6/2020 10:37:17 PM WET	12/7/2019 10:37:26 PM WET

Mail Box key store keypair generated by the PKI

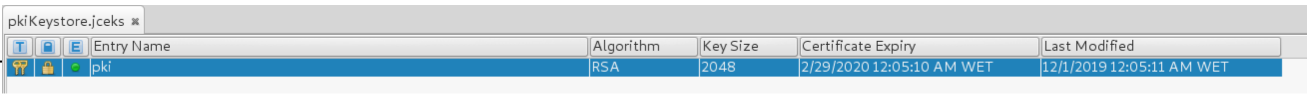


serverTruststore.jceks

T	E	Entry Name	Algorithm	Key Size	Certificate Expiry	Last Modified
-	●	pki	RSA	2048	2/29/2020 12:05:10 AM WET	12/7/2019 10:19:27 PM WET

Mail Box trust store with certificate signed by the PKI

PKI Server:



pkiKeystore.jceks

T	E	Entry Name	Algorithm	Key Size	Certificate Expiry	Last Modified
	●	pki	RSA	2048	2/29/2020 12:05:10 AM WET	12/1/2019 12:05:11 AM WET

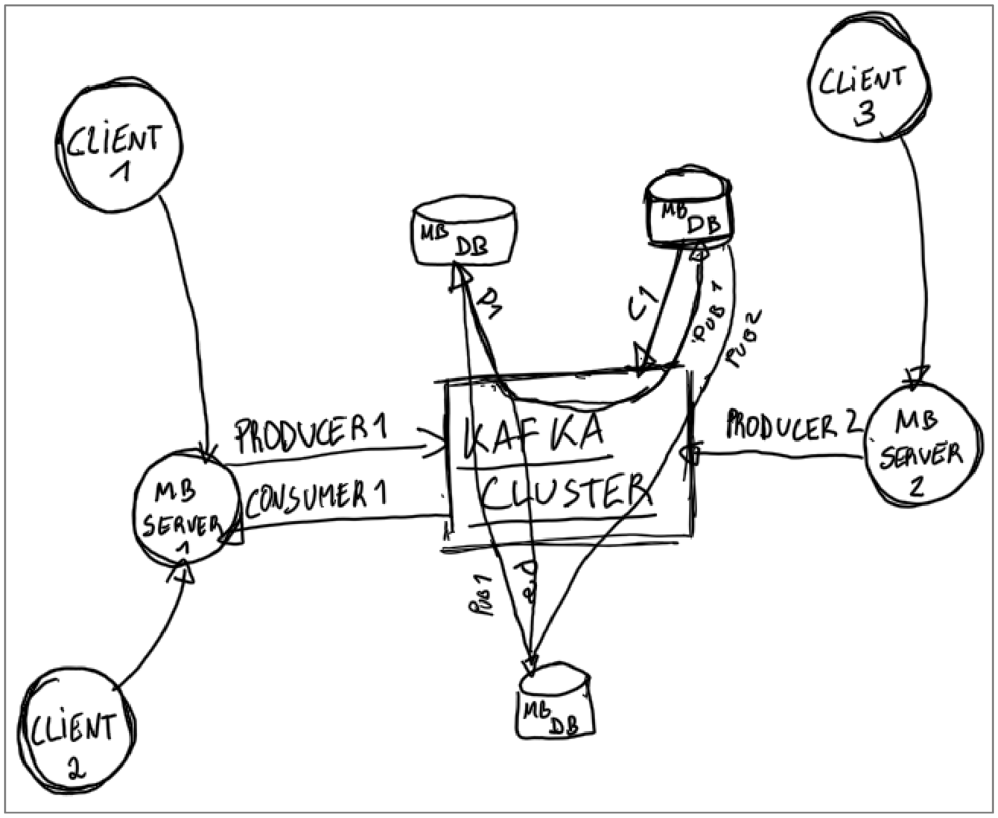
PKI keypair (Pub and Priv)

4. Work Evaluation and Validation

Although we did not replicate the databases, this would be something we could also do. For example, using a PUB/SUB system like Apache Kafka with Zookeeper making sure that every copy of the database has the most recent data and data. Alternatively, for example, an AWS service named "Amazon Aurora Global Database" that replicates data with no impact on the database performance and enables fast reads with low latency.

Doing this would allow having the services distributed into more places and supports more significant scalability.

For example:



Mailbox Server Kafka Cluster

The PKI would have a very similar architecture to this one, also resorting to a Kafka cluster.

As for the testing of the system, we tested with different configuration ciphers, such as:

- For SEA SPEC

  AES/CBC/PKCS5Padding 256 bits
  AES/ECB/PKCS5Padding 256 bits
  AES/CBC/NoPadding 256 bits
  Blowfish/CBC/ PKCS5Padding 448 bits
  DES/CBC/ PKCS5Padding 64 bits

- For MAC SPEC

  AES-GMAC 128 bits
  RC6-GMAC 256 bits
  RIPEMD-HMAC 256 bits
  SHA2-HMAC 512 bits
  AES-CMAC 256 bits
  DES-CMAC 56 bits

  For sea spec and mac spec, there are more options available, in the case of MAC's the MACHelper.class has a list of options available.
  We created a custom implementation that allows the user to use an HMac, CMac, or GMac construction.

- User UI hash

  SHA-256

- SEA Helper for DH file encryption:

    AES/GCM/NoPadding (key trimmed from users KS password)

- TLS Versions

    TLSv1.2

    TLSv1.1

- TLS CipherSpec

    TLS_RSA_WITH_AES_256_CBC_SHA256

    TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256

    TLS_DHE_RSA_WITH_AES_256_CBC_SHA256

    TLS_DHE_RSA_WITH_AES_128_CBC_SHA256

    TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA

5. Conclusion

Designing this system was a great challenge due to its open nature. All users have access to all details, meaning there is no actual access-control in the server itself. This means encryption and authentication schemes are extremely significant, and their strength will dictate the strength of the whole system. By using Diffie-Hellman public key exchanges, it made it possible to keep absolutely no keys in the server, making the encrypted data in the server, virtually useless for any attacker.

Another important aspect is the “democratization“ of the user’s cipher spec. This means that every user can choose how secure they want their data to be, with the respective overhead being processed on the client-side.

The TLS implementation allowed the client-server interactions to make use of a secure channel. Furthermore, by resorting to the PKI as a CA, the implementation became a lot richer with a simplified OCSP solution. The ability to revoke a certificate adds a lot more security to the system by revoking compromised keys. By using certificate chains, the server and client never really need to know each other; they only need to know the common trusted CA, which allows for much easier scalability.

Another challenge was the implementation of a system that would work with any cipher specification, which meant studying algorithms and the Bouncy Castle API. This process also had to be transparent to the user encrypting, and the user decrypting, which meant all the necessary adjustments were made on the fly by the custom helpers we developed.

This assignment made us apply all the knowledge we learned in the course to make a fully viable, reliable, and secure messaging system.