RSA-based Public-key Certification Authority (CA)

Mohammed Kaif & Arya Sinha *(2020084 & 2020498)*
CSE350: Network Security
Assignment 3 - Project 0 - README

Classes

The main file consists of 2 classes:

- Client
- Certificate Authority (CA)

Client

Variables

- **ID** - Unique ID of the client
- **p** - prime number 1 to generate keys
- **q** - prime number 2 to generate keys
- **CA_PU** - public key of the CA
- **certificates** - dictionary of issued certificates for this client
- **my_certificate** - client's own certificate
- **messages** - list of received messages
- **public_key, private_key**

Functions

- **get_request()** - Returns the encrypted request for a certificate for a specified user.
- **get_my_cert()** - Stores the client's own certificate.
- **add_cert()** - adds a new certificate to the client's certificates dictionary.
- **check_cert()** - Checks if the client has a certificate for a specified user.
- **get_cert()** - Returns a certificate for a specified user.
- **verify_cert()** - Verifies the validity of a certificate by checking the hash value against the decrypted hash in the certificate, and checking if the certificate is still valid.
- **show_certificates()** - Prints the client's certificates.
- **encrypt()** - Encrypts a message using a specified key
- **recieve_message()** - Decrypts a message using the client's private key and adds it to the messages list.
- **show_messages()** - Prints the client's received messages
- **check_validity()** - Checks if the client's own certificate is still valid

Certificate Authority (CA)

Variables

- **ID** - Unique ID of the CA
- **p** - prime number 1 to generate keys
- **q** - prime number 2 to generate keys
- **PUs** - dictionary of public keys for each user who has registered with the CA
- **certificates** - dictionary of issued certificates for each registered user
- **public_key, private_key**

Functions

- **set_PUs()** - Registers a new user's public key with the CA
- **get_certificate()** - Decrypts the request using the CA's private key and returns the corresponding certificate for the requested user, if one exists.
- **make_certificate()** - Decrypts the request using the CA's private key, generates a new certificate for the requested user, and stores the certificate in the certificates dictionary.

Assumptions

- That clients already (somehow) know their own [private-key, public-key], but do not have their own certificates or that of others,
- That clients already (somehow) know the public key of the certification authority,
- That CA has the public keys of all the clients.

RSA implementation

The simple public key infrastructure (PKI) uses an in-built RSA encryption from scratch. This RSA implementation in present in the helper *my_rsa.py* file. The main python file imports this for its RSA functionality.

Functions

- **generate_keypair(p, q)** - This function takes two prime numbers p and q as inputs and returns a tuple (public_key, private_key). The function first calculates the modulus n as the product of p and q, and then calculates the Carmichael's totient function (also known as the lambda function) of n as (p-1) * (q-1). It then generates a random number e between 2 and lambda_n (exclusive) such that e and lambda_n are coprime (have no common factor other than 1). It does this by repeatedly generating a new random number until e and lambda_n are coprime. It then calculates the modular multiplicative inverse of e modulo lambda_n using the *inverse_mod*() (explained below), which is the private key d. Finally, it returns the public key as (n, e) and the private key as (n, d).

- **inverse_mod(a, m)** - This function takes two integers a and m as inputs and returns the modular multiplicative inverse of a modulo m for co-prime numbers a and m. The function first checks if a and m are coprime (have no common factor other than 1). If not, it returns None. Otherwise, it uses the extended Euclidean algorithm to calculate the modular multiplicative inverse of a modulo m and returns it.
- **encrypt(pk, plaintext)** - This function takes a public key pk and a plaintext string as inputs and returns the corresponding ciphertext as a list of integers. The function first unpacks the public key into n and e. It then converts each character in the plaintext string to its Unicode code point, raises it to the power of e modulo n, and appends the result to a list. Finally, it returns the list of encrypted integers.
- **decrypt(pk, ciphertext)** - This function takes a private key pk and a ciphertext as a list of integers as inputs and returns the corresponding plaintext string. The function first unpacks the private key into n and d. It then raises each integer in the ciphertext to the power of d modulo n, converts the resulting integer back to its corresponding Unicode character, and appends it to a list. Finally, it joins the list of characters into a single string and returns it.

Hashing

Hashing is done using the SHA256 function imported from the **hashlib** python library

Code Explanation

This code implements a simple public key infrastructure (PKI) using RSA encryption for secure communication between clients. The PKI is based on a certificate authority (CA) that provides certificates to clients upon request. Each client has a public and private RSA key pair, and the CA has its own key pair.

The certificates are of the form -

CERTA = [IDA, IDC, PUA, TA, DUR, ENC(PR_CA, Hash(IDA, IDC, PUA, TA, DUR))]

where

- IDA is user ID
- IDC is information (ID) about certification authority
- PUA is public key of A
- TA is time of issuance of certificate - obtained using **time.time()** (pre-existing python library)
- DUR is the duration for which the certificate is valid
- ENC is the encryption algorithm used by CA
- PRCA is public key of certification authority

The hashing function used in SHA-256.

The code defines two classes: **CA** and **client**. The CA class has methods create certificates, store public_keys of clients and return certificates. The client class has methods for sending and receiving messages, requesting certificates from the CA, and verifying the certificates. The code includes a while loop that prompts the user to create new clients, log in as an existing client, and perform various tasks, such as requesting certificates and sending messages.
The RSA scratch implementation is present in the helper *my_rsa.py* file imported as rsa in the main file.
2 clients have been made with IDs 1 and 2, but new clients can always be added. Every client must first request for its own certificate from the CA for the CA to recognize the client and create a sharable certificate for it. Every certificate expires **DUR** seconds (60s in this case) after its creation, therefore a client must keep updating its certificate to receive new messages.