PKDA (Public Key Distribution Authority)

PKDA is a centralized authority which validates certificates and provides public keys for users on demand. This implementation of PKDA supports providing users with the public key of requested users and provides a technique to validate provided public keys.

How it works

- PKDA first loads the user_dataset.json into the database (RAM) and waits for clients to send requests.
- Client prepares a JSON request of the following kind.

  ```
  {
      "service": "request-public-key",
      "id-initiater": "<id>",
      "id-responder": "<id>",
      "time": "<time>",
      "nonce": "<nonce>"
  }
  ```

  Here, `id-initiator` refers to the ID for the user who is requesting this service, `id-requested` refers to the ID of the user whose public key is requested, `time` is current time in milliseconds from epoch for preventing reply attacks, and lastly, `nonce` is a generated string for validating the response from the PKDA. For example a valid request looks like so:

  ```
  {
      "service": "request-public-key",
      "id-initiator": "0",
      "id-requested": "1",
      "time": 1711713195,
  ```

```
    "nonce": "the one who lays in dormant..."
}
```

- Client encrypts the request with `PUBLIC_KEY` (e, n) of PKDA.
- PKDA decrypts the request with its `PRIVATE_KEY` (d, n), then proceeds to process and validate the request. After validation is done, it extracts the user's `PUBLIC_KEY` from the database and prepares a response of the following kind.

  ```
  {
      "service": "request-public-key",
      "id-requested": "<id-requested>",
      "e": "<publickey>",
      "n": "<modulus>",
      "nonce": "<nonce>"
  }
  ```

  Again, here, `id-requested` refers to the ID of the user whose public key is requested, `e` & `n` are the public key of the requested user. Lastly, the `nonce` is the same as in the request, which the user can compare to verify that no tempering is happening on the response sent by the PKDA. For reference a sample response looks like so:

  ```
  {
      "service": "request-public-key",
      "id-requested": "1",
      "e": "65537",
      "n": "3413536679262404565245364\u200b62...continued",
      "nonce": "the one who lays in dormant..."
  }
  ```

- The response is encrypted with the `PRIVATE_KEY` of PKDA and sent to the user. The user then proceeds to process and validate the response by comparing the nonces. If the nonces are the same, that indicates the PKDA was adequately able to decrypt and process the request, and no tempering happened on the response, as it would have led to the change in the nonce received after decrypting the PKDA response.

> There is no need to encrypt the response via `PUBLIC_KEY` of the initiator, as that would result in confidentiality, which is unnecessary as all the information is public for anyone to access anyway.

RSA (Rivest–Shamir–Adleman)

> All the asymmetric encryption and decryption in PKDA are done via RSA.

As RSA requires the data to be in integer representation. Thus a custom encoding was needed to be developed. The preprocessing is as follows:

- First the string is chunked into multiple consecutive substrings; as RSA requires the message to be less than `n` (modulus). String is chunked in such a way that every substring is of the same length and all the substring's integer representation is less than `n`. The definition of `chunk_size` can be found here.
- After the chunking, each substring is encoded into integer representation. Each character is encoded via 2 digit number and concatenated to form the integer representation. The encoding can be found here.

Encryption of each substring happens independently of each other and other connected via a counter mode (Galois) just like the one in AES/GCM algorithm. So that two substrings containing same text should not lead to same encryption.

At the end of substring encryption the resultant is padded with leading zeroes to make it of same length as the `n`. Such that after concatenation of all the substrings; a border can be drawn between substrings for decryption.

> Decryption happens in same way but in reverse.
>
> Implementation of RSA can be found here.

> Note: Using incorrect keys for encryption and decryption may lead to errors, as the implementation heavily relies on the encoding and using incorrect keys will break the encoding.

Guide

- **Generate keys**: This will generate keys of specified key size (default-2048) and print the relevant information to the console (n, e: encryption key, and d: decryption key) also store the keys in JSON file (keys.json).

  ```
  ./rsa genkeys <key_size: optional; default-2048>
  ```

- **Encryption**: This will encrypt the provided text with the provided key.

  ```
  ./rsa encrypt "<text>" <e> <n>
  ```

- **Decryption**: This will decrypt the provided encrypted text with the provided key.

  ```
  ./rsa decrypt "<encrypted-text>" <d> <n>
  ```

Quick start

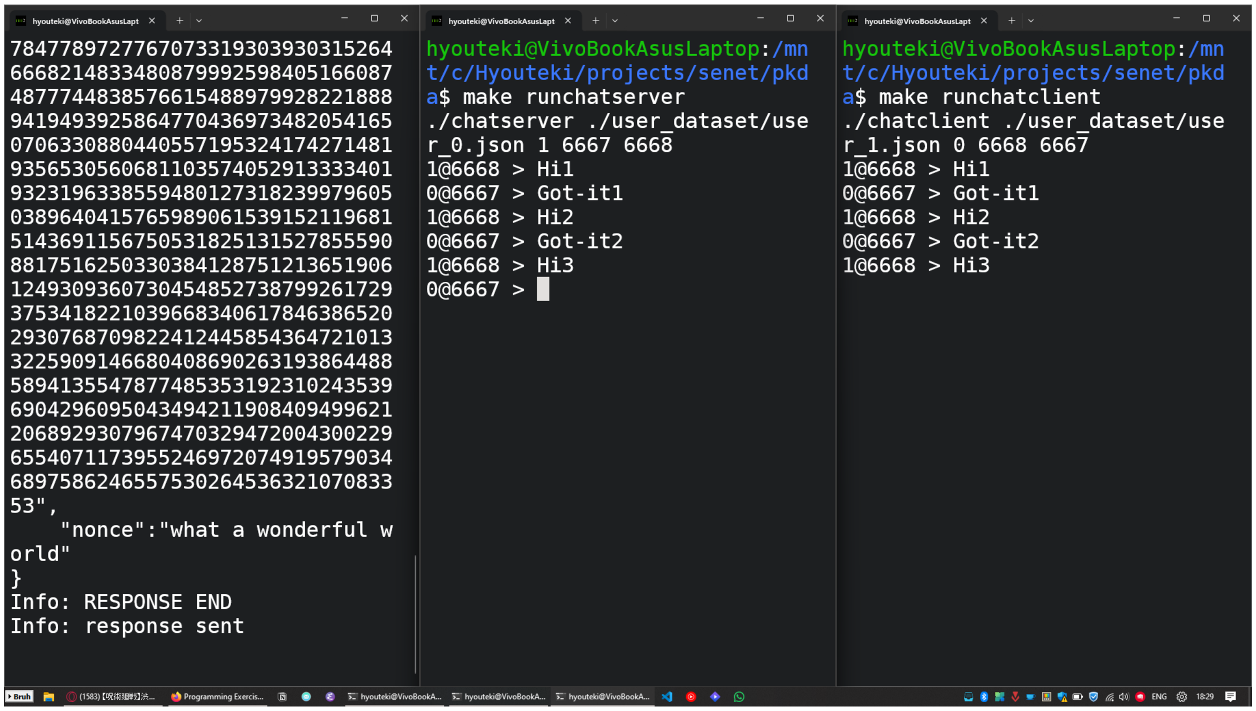
- Install the required packages using `make install`.
- Build the project using `make`.

Guide for PKDA

Run the PKDA using `./runpkda` binary required that the project is already built.

Application

A simple encrypted chat client is made. Two users chatserver.c and chatclient.c chats with each other. Chat is initiated by the `chatclient`. Both the users do not know each others public key for encryption and decryption. Thus they make requests with PKDA and receive each others public keys and proceeds with the chatting.



```
7847789727767073319303930315264
6668214833480879992598405166087
4877744838576615488979928221888
9419493925864770436973482054165
0706330880440557195324174271481
9356530560681103574052913333401
9323196338559480127318239979605
0389640415765989061539152119681
5143691156750531825131527855590
8817516250330384128751213651906
1249309360730454852738799261729
3753418221039668340617846386520
2930768709822412445854364721013
3225909146680408690263193864488
5894135547877485353192310243539
6904296095043494211908409499621
2068929307967470329472004300229
6554071173955246972074919579034
6897586246557530264536321070833
53",
    "nonce":"what a wonderful w
orld"
}
Info: RESPONSE END
Info: response sent
```

```
hyouteki@VivoBookAsusLaptop:/mn
t/c/Hyouteki/projects/senet/pkd
a$ make runchatserver
./chatserver ./user_dataset/use
r_0.json 1 6667 6668
1@6668 > Hi1
0@6667 > Got-it1
1@6668 > Hi2
0@6667 > Got-it2
1@6668 > Hi3
0@6667 >
```

```
hyouteki@VivoBookAsusLaptop:/mn
t/c/Hyouteki/projects/senet/pkd
a$ make runchatclient
./chatclient ./user_dataset/use
r_1.json 0 6668 6667
1@6668 > Hi1
0@6667 > Got-it1
1@6668 > Hi2
0@6667 > Got-it2
1@6668 > Hi3
```

Guide

- Open three separate terminals in this directory.
    - Launch PKDA on one using `./runpkda`.
    - Launch chatserver on another using `make runchatserver`.
    - And lastly run `make runchatclient` on the last terminal.
    - Voila!! you are done.

Members

- Lakshay Chauhan (2021060)
- Ankit Kumar (2021015)