Documentation

Requirements

This project was primarily developed using Python 3.6.9 and Ubuntu 18.04 LTS and Windows 10

Python 3.6+

- We make use of Python 3.6+ f-Strings which are not available in prior versions. A workaround is to convert all instances of f-Strings to their string format equivalent, though that may be a tedious process
- We make use of Python 3's builtin `pow(x, y, z)`, which is only available in Python 2 as `math.pow(x, y, z)`
- We also make use of Python 3's prepackaged `secrets` module for generating random numbers in two places, which is not available in Python 2 by default
    - Note that we did not feel the implementation of our own RNG neccessary. We make use of the RNG very few times, meaning that the OS can easily provide more than enough random bits from `\dev\random` to satisfy our needs if we decided to implement the `randbits()` functionality ourself
- We also make use of Python 3's prepackaged polling library `select` on the server side to handle multiple concurrent sockets. The wrapper should abstract away OS level details, but if there are any networking issues, a Linux machine, VM, or even WSL 2 will likely solve your issues

Quick Start

1. Navigate to the project's root directory
2. Start the server
    - `python3 server.py`
3. Open another terminal to the same directory
4. Start a client
    - `python3 client.py`
5. You may start more clients in other terminals as well if you want
6. Make use of the following (self explanatory) commands in any client terminal
    - login
    - logout
    - quit
    - help
    - echo
    - show-balance
    - withdraw
    - deposit
7. There are several test accounts you may login as

```
{
    "samarth": {
        "password": "woodcuriousblankpossible",
        "balance": 4.20
    },
    "owen": {
        "password": "centraltaskpureexchange",
        "balance": 10101.01
    },
    "max": {
        "password": "streetcaraboardcombination",
        "balance": 54321
    }
}
```

Repository Structure

In efforts to make working with our code easier, we have created an outline of our code structure.

- `crypto\`
    - `AES.py`: implementation of AES-256 symmetric encryption
    - `SHA1.py`: implementation of SHA1 hash algorithm
    - `DSA.py`: implementation of DSA algorithm
- `utils\`:
    - `BankAccounts.py`: implements a mock database used for storing and accessing banking information
    - `keys.py`: exports cryptographic keys used by the client and server. Note that in practice, the secret keys in this file (which are expressly noted as such) would be securely loaded at runtime and not hard coded as such
    - `messages.py`: exports string constants used during the handshake and functions used to serialize / deserialize messages
    - `numbers.py`: exports functions for deserializing floats and ints
    - `symmetric_encryption.py`: wraps our implementation of AES-256 to encrypt and decrypt messages
    - `digital_signature.py`: wraps our implementation of DSA to sign and verify messages
- `client.py`: the client side implementation of our protocol
- `server.py`: the server side implementation of our protocol

Communication Protocol Overview

The following is an overview of how communications are securely established and sent over a (theoretically) insecure TCP connection

Initial Connection

1. The bank (henceforth referred to as Server) constantly awaits ATMs to connect via a TCP socket on a publicly published port
2. An ATM (henceforth referred to as Client) connects to the bank using the bank's domain name and published TCP port number
3. The Client retrieves the Server's public signature key from either a Certificate Authority or, more likely, an immutable hard-coded value
    - Every message received from the Server is expected to be in the form `message|signature`, where the signature is a DSA signature applied to the SHA1 hash of the message
    - Encryption, whenever used, is to be applied on top this signed message following the "Sign, then encrypt" methodology
    - Receiving an unsigned or unsuccessfully verified message automatically closes the connection
    - Messages from the client to the server do not have any signature

Key Exchange (Creating a Secure Channel)

1. Client waits for the server to send the message `OK|START_KEY_EXCHANGE`
2. Client uses published Diffie-Hellman group values to generate it's value `A` and sends it to the server in the format `OK|DH1[A...]`
3. Server receives this, deserializes it, and calculates it's own value `B` and sends it the Client using the same format
4. Both the Server and Client perform the final Diffie-Hellman calculation to arrive at their shared key
5. To confirm that both Server and Client arrived at the same key, they send both send a test message
    - The Client first sends `OK|request to start session`, AES-256 encrypted with their shared key
    - The Server receives this message, decrypts it to verify that it is the expected message, and sends `OK|yes please start session` encrypted using the key back to the Client
    - If either the Server or the Client receives an unexpected message during this process they (as in the party who received the unexpected message), should stop communicating and close the connection.
    - As noted earlier, this server to client message is signed first, and then encrypted

Sessions (Communicating over the Secure Channel)

- All future messages between the Server and Client must be encrypted
    - The sender should take the intended message (referred to as message body), create a SHA1 hash of it, and encrypt `message|hash`. In the case of server to client messages

    where a signature is required, the message is effectively `original_message|sha1+DSA_of_original_message|SHA1_of_everything_preceding`
  - This is mainly to create standardized procedures for handling any combination of encryption and verification during the sending / receiving process
- The Client is now free to send requests to the Server
  - All requests are in the form `request_type|username.password|message_no|request_specific_arguments`
  - The credentials are verified by the bank and the request is immediately rejected if they do not match
    - Internally, all passwords are stored as the SHA1 hash of `128-bit_salt|password`
    - Verifying credentials takes somewhere randomly between 0.1 and 0.6 seconds at minimum, so an average of 3 passwords a second can be tested by an attacker
    - Additional safeguards such as maximum retry count etc. can be implemented as well, but that is outside the scope of this project (should be implemented at the Bank API level)
  - The `message_no` starts at 0 and is incremented every time a message is sent (either direction). If the message_no does not match the message_no expected by the server or client, an error message is sent and the connection is immediately closed
  - Requests abiding by these two constraints (assuming valid formatting) are processed and an appropriate response for the request is sent back in encrypted form

Protocol Security Analysis

Shared Key Generation

The crux of our security is derived from generating a shared key between the Client and the Server. This shared key, when used with AES-256, provides a way for the server and client to communicate with a high degree of privacy and confidence in each other's authenticity. More on the benefits of symmetric encryption in our AES discussion as well as in the next section.

Our method of generating this 256 bit key comes from the classical (not elliptic curve) Diffie-Hellman procedure. In particular, we use Diffie Hellman Group 16, a 4096 bit group published by the IETF

An alternative to this process we considered was to have the Client encrypt a session key using the Server's public key, which would make it so only individuals with the Server's private key (namely the Server itself), would be able to decipher it.

However, we opted against this in favor of Diffie-Hellman for the sake of `perfect forward secrecy`. In other words, we wanted to make it so that future compromises of our system (whether client side or server side) would not leak any information about past transactions. The proposed PKC scheme mentioned has a vulnerability in the case where an attacker gains access to the Server's private key. In this case, all past communications with the server, had they been intercepted and recorded, could be easily decrypted by simply decrypting the original key exchange message the Client sent to the Server. Our Diffie-Hellman, and all Diffie-Hellman schemes for that matter, do not suffer from such vulnerabilities since an attacker cannot recover the secret random used by the server in the same way.

At the end of this exchange, we have arrived at a key that we can safely assume is only known to the Client and Server

Communication over the Secure Channel

Symmetric encryption offers us an avenue to now produce a secure channel through which we can ensure the authenticity of the Client and the Server

It is safe for us to assume this because:

- We define a client simply as the one in possession of the Diffie-Hellman secret random unknown to the Server
    - Due to the properties of AES, it is extremely hard someone not in possession of the key to create a message, which, when decrypted with the key, decrypts to a valid message
    - The server sends all messages signed using their private key, so we can ensure the message, at some point, originated from the Server

Once was ensure authenticity of the message's origin, we need to ensure the message has not been corrupted (maliciously or otherwise)

- We include a message hash for this purpose

We also need to ensure the message has not already been used before to prevent replay attacks

- Therefore, we introduce the `message_no` field to ensure each message can be used at most one time

With all these techniques in place, we firmly believe our protocol is secure

Footnote

Our bank uses floating point arithmetic for banking operations, which is not recommended by any means. In reality, the bank would use something like BigInt and other arbitrary precision arithmetic libraries. An alternative would be to represent the amount in the smallest denomination possible (cents). Since such things are outside of the scope of our cryptography project, we did not use them to keep or Banking API as minimal as possible.