CSCI4230 Project Documentation

Aidan McHugh, Kai Orita, and Bradley Presier

April 2023

GitHub

Note: If links do not work, try opening this file from its directory using Edge or Chrome.

1 Overview

This project implements a bank ATM system, allowing users to deposit, withdraw, and view their balance through an ATM client. Due to the lack of a physical machine, no real cash can be exchanged, but we assume for our purposes that this occurs if the ATM client is not an impostor.

The implementation operates on three levels with different protocols, which are

1. TCP - enabling a connection between client and server
2. TLS/SSL - authenticating and encrypting communication between client and server
3. Application - authenticating the user and providing ATM functionality

Note that we consider the server, client, and user as three parties who must each be authenticated at some level. Also note that the TLS/SSL and Application levels each have their own mode of error handling, which will be discussed in their respective sections.

2 Usage

2.1 Setup

This project requires Python 3, and has not been tested with versions earlier than Python 3.10.0. We compile and build local Python packages. This should not be an issue in most Python 3 installations.

To install dependencies,

1. (optional) Create a virtual environment of your choice.
2. In this directory, run `pip install -r requirements.txt`

2.2 Run Application

1. Run `python server.py` to start the server.
2. Run `python client.py` to start a client instance.

2.3 Provided Accounts

We have provided several accounts in the database that team blackhat might use or attack:

Mallory Malificent

This is you!

Card Number	`0000000000000000`
CVC	`666`
Expiration Date	`04/2025`
PIN	`6969`

Charlie Collaborator

If you steal from Charlie you're a bad friend. But maybe they'll let you intercept their messages for science.

Card Number	`0000000000000505`
CVC	`111`
Expiration Date	`05/2025`
PIN	`1111`

Alice Allison

Alice keeps her bank information very secret.

Card Number	(random)
CVC	(random)
Expiration Date	`05/2025`
PIN	(random)

Bobby McBobface

Bobby is less good at keeping secrets than Alice.

Card Number	`0505050505050505`
CVC	`123`
Expiration Date	`06/2023`
PIN	(random)

Victor Evilson

Maybe you feel bad about stealing from Bobby. Victor is very evil so the only issue with stealing from him is that he might come find you.

Card Number	`4111111111111111`
CVC	(random)
Expiration Date	`09/2026`
PIN	(random)

Billy Bazillionaire

Billy has a lot of money. He probably wouldn't miss it if some disappeared, right?

Card Number	(random)
CVC	(random)
Expiration Date	`12/2100`
PIN	(random)

2.4 Structure Overview

This project is implemented using Python 3 and C++. Specifically, Python is primarily used while an included Python package is implemented using C++. Throughout this project, we have tried to be thorough in In the following, we will describe the different components:

2.4.1 Client/Server

The client (`client.py`) and server (`server.py` and `database.py`) exist in the main directory of this project. These are the main entrypoints for this project, and neither party should have access to the internal memory of the other process.

2.4.2 Shared Libraries

These libraries are contained in `./shared/` and are used by both the server and client (but do not share data between the two). The libraries in this directory are implemented in Python.

2.4.3 C++ library

This Python module is implemented using C++ in `./lib/` and will be built by `pip` when installing dependencies.

2.5 Order of Operations

The operation of this system can be categorized into three primary phases, in addition to the establishment of the TCP connection.

1. (TCP Layer) TCP connection established
   Client: Connection via `socket`
   Server: Connection via `socketserver` using `server.Handler`

2. (TLS Layer) TLS/SSL handshake establishes `shared.rpi_ssl.Session`
   Client: Buffer control given to `shared.handshake_handler.client_handle_handshake`
   Server: Buffer control given to `shared.handshake_handler.server_handle_handshake`

3. (App Layer) Authenticate user to associate `database.Account` with session
   Client: Get `shared.card.Card` information from user
   Server: Verify information in `database`

4. (App Layer) Accept routine ATM commands
   Client: Get command details and send request to server
   Server: Respond to command as appropriate

3 TLS/SSL Implementation

At this level, we attempt to implement a simplified version of TLS 1.3. In this, we attempt to implement cryptographic algorithms in a manner that is usage-independent and reusable.

3.1 Symmetric Encryption

We implement only AES-256 with Cipher Block Chaining. This is contained in the C++ library. This differs from TLS 1.3 in that CBC-mode is not permitted by TLS 1.3, but is otherwise compliant.

3.2 Hash Functions

We implement SHA-1, SHA-256, and SHA-384 in `shared.rpi_hash`. However, in keeping with TLS 1.3 and best practices, SHA-1 is not considered a valid option for any purpose.

3.3 Message Authentication Codes (MACs)

We implement HMAC in `shared.rpi_hash`.

3.4 Digital Signatures

We implement only RSASSA-PSS (Signature Scheme with Appendix; Probabilistic Signature Scheme) in `shared.rsassa_pss` as specified in RFC 8017. This is a preferred digital signature scheme by TLS 1.3 and produces a non-deterministic signature that can be used to verify the original message using the RSA public key. Our implementation can apply any hash function, but we use SHA-256 and SHA-384 in keeping with TLS 1.3 and best practices. Both `rsa_pss_rsae` and `rsa_pss_pss` modes as defined by TLS 1.3 use this algorithm, **TODO: DO WE SUPPORT BOTH OR JUST ONE?**

3.5 TLS Records

Each message in a TLS/SSL session is contained within a TLS record. Functions to encode and decode these, along with a `Session` class containing negotiated TLS/SSL session information are implemented in `shared.rpi_ssl`. A `Session` will only exist after the handshake is completed, and thus the encrypted Alert and Application record implementations are included within it. The `Session` can automatically manage encryption, decryption, and verification/creation of MAC. Essentially, it provides an interface for the Application-level.

3.6 TLS/SSL Error Handling

Should a TLS/SSL-related error occur on this level, an `SSLError` will the thrown, which should be reported to the other party via a corresponding TLS Alert record. As specified by TLS 1.3, error alerts (all alerts except `close_notify` and `user_canceled`) must be considered fatal.

4 Application-Level Protocol

This is also partially described in `shared.protocol`. All messages at the application level should be in the body of an SSL/TLS application record. This means that application level messages are encrypted and authenticated between the client and server (but not necessarily the user). Note messages sent follow the format of a client request, followed by a server response. The server should never spontaneously send a message.

4.1 Application Message Format

Each application-level message starts with a single header byte describing what application message type it contains. These types are defined by the enum `shared.protocol.MsgType`. The rest of the message contains further details:

4.1.1 `ACCOUNT_AUTH` message type

This message type is used when authenticating the user. Prior to user authentication being completed, only `ACCOUNT_AUTH` and possibly `ERROR` messages should be sent. If the server responds successful, user authentication is now completed and the session is permanently associated with the provided account.

Request Format

Start	Length	Content	
0x00	0x01	`0x00`	(`ACCOUNT_AUTH` header byte)
0x01	0x0c	Card Data	(formatted with `Card::to_bytes()`)

Response Format

Start	Length	Content	
`0x00`	`0x01`	`0x00`	(`ACCOUNT_AUTH` header byte)
`0x01`	`0x01`	`0x00` or `0x01`	(unsuccessful or successful, respectively)

or, if attempts have been exceeded, the response is a fatal `ATTEMPTS_EXCEEDED` error message$_{\downarrow}$ (this can be considered unsuccessful).

4.1.2 `BALANCE` message type

This message type should only be used after the user is authenticated. It requests the current account balance of the user from the server.

Request Format

Start	Length	Content	
`0x00`	`0x01`	`0x01`	(`BALANCE` header byte)

Response Format

Start	Length	Content	
`0x00`	`0x01`	`0x01`	(`BALANCE` header byte)
`0x01`	`0x08`	big-endian unsigned integer	(account balance in cents)

4.1.3 `DEPOSIT` message type

This message type should only be used after the user is authenticated. It requests that an amount be added to the user's balance. With a valid ATM client, the corresponding amount of cash will have been inserted. While `DEPOSIT` returns its success status in a similar manner to `WITHDRAW`, it will rarely if ever be unsuccessful.

Request Format

Start	Length	Content	
`0x00`	`0x01`	`0x02`	(`DEPOSIT` header byte)
`0x01`	`0x08`	big-endian unsigned integer	(deposit amount in cents)

Response Format

Start	Length	Content	
`0x00`	`0x01`	`0x02`	(`DEPOSIT` header byte)
`0x01`	`0x01`	`0x00` or `0x01`	(unsuccessful or successful, respectively)

4.1.4 `WITHDRAW` message type

This message type should only be used after the user is authenticated. It requests that an amount be deducted from the user's balance. With a valid ATM client, the corresponding amount of cash will be provided if successful.

Request Format

Start	Length	Content	
`0x00`	`0x01`	`0x03`	(`WITHDRAW` header byte)
`0x01`	`0x08`	big-endian unsigned integer	(withdraw amount in cents)

Response Format

Start	Length	Content	
`0x00`	`0x01`	`0x03`	(`WITHDRAW` header byte)
`0x01`	`0x01`	`0x00` or `0x01`	(unsuccessful or successful, respectively)

4.1.5 `ERROR` message type

This message type represents a serious error with a request at the application level. It should only be sent by the server as a response to a client request. Error codes are defined by the enum `AppError` in `shared.protocol`. See the next section↓ for error type details.

Response Format

Start	Length	Content	
0x00	0x01	`0xFF`	(`ERROR`)
0x01	0x01	error code	(from `AppError` header byte)

4.2 Application Error Codes

Application error codes may be sent by the server in an `ERROR`-type application message in response to client requests. They should not occur or be sent in any other situation. If a valid response exists for the request's own message type (for example insufficient funds for `WITHDRAW`), then that will be used instead.

4.2.1 `INVALID_STAGE` (`0x00`)

This error code may be sent when a application message is sent at an improper time. For example, if `ACCOUNT_AUTH` messages are sent after the user is authenticated, or if any `BALANCE`, `DEPOSIT`, or `WITHDRAW` messages are sent before a user is authenticated. Additionally, this error code may be sent if a nonexistant message type is received.

4.2.2 `BAD_MESSAGE` (0x01)

This error code may be sent when the content following a valid error code appears to be improperly formatted. Note that as an application-level error, this will not occur if SSL/TLS issues arise.

4.2.3 `ATTEMPTS_EXCEEDED` (0x02)

This error will occur if the session has exceeded its permitted failed user authentication attempts or if the specified card number has recieved too many recent failed login attempts. As such, it can only occur in response to an `ACCOUNT_AUTH` request. This error is always fatal. For more details, see below↓

4.3 Credit/Debit Cards

Implementation for card formats and related details is in `shared.card`. The `Card` class represents the fixed-length fields of a standard credit or debit card:

Card Number	A 16-character numerical string which passes the Luhn test.
CVC	An integer in the range `[000,999]`.
Month	An integer in the range `[1,12]`.
Year	An integer in the range `[2000,3023]`.
PIN	An integer in the range `[0000,9999]`.

These can be serialized to and from 12 bytes for transmission.
Additionally, a `Card::generateRandom()` method is provided which will generate a random valid card, optionally with some set fields.

4.4 Database

Implementation for the `database` relies heavily on the above `shared.card.Card` implementation. Primarily, the database is made up of `database.Account` objects, which store information associated with the account, including the card, balance, and accountholder's name. Note that the name is not used for verification.
The database implemented here is intended to represent an abstraction of a real database, but does not fully implement the features of a typical database. For example, restarting the server will cause a database reset, as all data is stored exclusively in memory.

The database exposes the `get_account()` function to the server (and only the server), and the server is permitted to modify a returned `Account`'s balance. The client and user may not access the database except through the server.

4.4.1 Login Attempt Limit

The database implements a limit on recent attempts to login with any card number. If 5 or more failed attempts have occurred in the past 30 minutes (including any attempts that failed because of this), the database will throw an `AttemptsExceededError` that should be caught by the caller and transmitted to the client.
Note that to prevent this being used as a method to find other users' card numbers, login attempts for unused card numbers will have similar behavior in this regard as login attempts providing invalid verification details. So just as five attempts with a correct card number but incorrect PIN will cause further attempts with this card number to error, so too will five attempts with an incorrect card number. This error is fatal.

4.4.2 Generating Accounts

A number of accounts are generated when the database is started. Many of these randomize their card fields. See above↑ for the known account details.