Build It Document

Protocol overview:

Our protocol involves sending encrypted messages back and forth between the ATM and the Bank. Messages are encrypted using AES-256 in CBC mode with a randomly generated 256-bit symmetric key and a randomly generated 128-bit IV. The IV will be generated for every message to ensure that they aren't predictable and the chances of repeat IVs will be extremely low. The Bank and ATM will keep track of certain internal states throughout the session, ensuring that whatever message each expects to receive as a reply is valid. Each message is time stamped with seconds and microseconds since the Epoch (00:00:00 1 January 1970). Both the ATM and Bank will constantly update their internal time stamps to ensure messages received are sent chronologically after the previous recorded time stamps. The ATM will also depend on a *.card file that the Bank issues for each account with the username of the account in place of the asterisk. The cards contain a unique string of 32 hex characters for every account. This information is sent along with every message, except the begin-session request and response, to establish that it's the same user sending requests from the ATM. For the begin-session request, only the username of the account is sent to verify if the username is indeed a valid account; the message will still include the time stamp just the card number is not used yet until after the bank confirms that the user does exist. In order for user to successfully login, they would need to provide a valid username, a valid card with a card number that correctly corresponds to the provided username, and the correct PIN for that account. Both the ATM and Bank will temporarily store the username and card number as part of their session data.

Account data in the Bank is stored as two hash tables. One will map card numbers to the corresponding username of that account. The other hash table will map the username to a struct called `Userfile` which stores the balance of that account as an int, PIN as a char array, and the card number as a char array.

Security Features:

Sanitizing Input

We ensured that the ATM and Bank are rigorous in sanitizing inputs and flushing out extra input. All buffers are set to a size of `DATASIZE` (sometimes `DATASIZE + 1`), which is set to 1000 currently, and we only accept up to a `DATASIZE` number of bytes through `stdin`. Any extra data beyond that will be completely discarded, as `DATASIZE` is guaranteed to be large enough to hold all valid data entries. All usernames will be sanitized to ensure all characters are alphabet letters and they do not surpass 250 characters. PINs are stored as 0-9 digit characters rather than a number value. Moreover, all actual number values will be sanitized to fit within an int. We also ensured unconventional inputs such as `0000000000000001` will be sanitized to be just `1`. Users can input various amounts of whitespace characters in between the arguments; those will also be sanitized out. In short, going over the `DATASIZE` with large amounts of whitespaces and leading zeros will render your input invalid.

Encryption and Decryption

As mentioned before encryption and decryption are done using AES-256 in CBC mode with a symmetric key and randomly generated IVs that are generated each time a message needs to be encrypted. This mode was chosen for its simplicity. Since AES-256 is already resistant to known plaintext attacks, as long as IVs aren't predictable, we felt it was secure to use CBC as long as we could generate random IVs for each encryption. Library functions used are provided by OpenSSL 1.1.1.

Recording Time Stamps

As mentioned before, the ATM and Bank will keep track of time stamps of messages. Each time stamp is created by calling `gettimeofday()` and recording the seconds and microseconds received from that function call. When ATM and Bank are first created, they will record the Epoch time into their internal state. Then for every message sent, both will record the time stamp of that message into their internal state. For cases where a message is received but no reply is due, ATM and Bank will record the time stamp of that received message instead. And for every message received, ATM and Bank will both check the time stamp to make sure that the message received took place after the latest message time recorded in their internal state. This will prevent replay attacks as a valid message will always have a larger seconds value. If not, then it will most definitely have a larger microseconds value. We assume that no computer can be fast enough to do all the necessary processing where valid sent messages and received messages have the same exact time stamp. This might change in the future, but given

that user input is also required between some ATM to Bank communications, this assumption should be safe for now.

Card Number Generation

Card numbers are generated by taking a string in this format: `<username>.card<pin>`. We then encrypt it using the symmetric key and a randomly generated IV. This IV value will be tossed as it becomes useless. We then hash the resulting ciphertext using MD5. This will give us 16 raw bytes that we then translate into 32 hex characters. The card generation function is written so that if a card number already exists for some account, it will re-roll until we get a card number that is unused. This removes any issues we have with collision. Despite knowing that MD5 is no longer cryptographically secure to create checksums, we aren't using it for that purpose in our system. It's just a way to generate 32 bytes of hex characters and we re-roll the hash if somehow we have a collision. Lastly, since the ciphertext is encrypted with our key and a secret randomly generated IV, this prevents the ability for any attacker to recreate the card number of an account despite knowing the username and PIN.

Message Structure

Structure of an initial user verification message: IV (16 bytes) + Epoch time (seconds, 8 bytes) + Epoch time (microseconds, 4 bytes) + **command (bolded)**

3 possibilities:

- **<username>**
- **no-user-found**
- **user-found**

Structure of a general message: IV (16 bytes) + Epoch time (seconds, 8 bytes) + Epoch time (microseconds, 4 bytes) + account/card number (32 bytes) + **command (bolded)**

11 possibilities:

- **unverifiable**
- **verify XXXX**
- **access-denied**
- **access-granted**

- **withdraw\0(<int> four bytes)**
- **dispense\0(<int> four bytes)**
- **dispensed\0(<int> four bytes)**
- **insufficient**
- **balance**
- **balance\0(<int> four bytes)**
- **end-session**

Protocol in Detail

> ❗ An alternate representation of the protocol in the form of a textual diagram can be found at the end of this document

Init:

The init program is written as a script that, when run, creates the following files, where <path>/<init-fname> is the path/filename specified by the user as a command line argument:

- <path>/<init-fname>.bank
  - This file contains a 32 byte symmetric key for AES-256, in the form of raw bytes, using a cryptographically secure pseudorandom number generator (OpenSSL)
- <path>/<init-fname>.atm
  - This file is a mere copy of <path>/<init-fname>.bank

User authentication and login:

> ❕ At this point, no messages contain a card number encrypted, as there is not yet a verified account

Once all the necessary components (init, router, bank, atm) are run, a user authenticates by beginning a session. The ATM and Bank start in an initial state. Upon receiving a command from `stdin` to start a session, the ATM sends the inputted username provided to the Bank and changes state to indicate it is waiting for user confirmation. Upon receiving the username, the Bank searches for the user in its records. If the user is not found, the Bank sends a message indicating that it could not find the user. Consequently, the ATM changes back to initial state and the Bank's state remains constant. If the user IS found, the Bank sends a message indicating that the user was found, changes state to waiting for a PIN, and stores the username as an active user. Upon receiving the Bank's message, the ATM stores the card number as the active card and reads the PIN from `stdin`.

> ❕ At this point, all messages sent will have the card number encrypted

If the PIN has incorrect format, the ATM sends a message indicating that the PIN is unverifiable. In this case, the ATM would clear the active card and change back to initial state. Upon receiving the ATM's message, the Bank clears the active user and changes back to initial state.

If the PIN *does* have correct format, the ATM sends a message to the Bank to request verification of that PIN. The ATM also changes state to indicate it is waiting for PIN verification. Upon receiving the ATM's message, the Bank stores the card number as the active card and checks stored PIN corresponding to the user. If the provided PIN is not equal to the stored PIN, the Bank sends a message indicating access is denied. In this case, the Bank clears the active card and changes back to initial state. Upon receiving the Bank's message, the ATM also clears the active card and changes back to initial state.

If the PIN values *are* equal, the Bank sends a message indicating access is granted. In this case, the Bank changes state to open session, and saves a pointer to the logged user's file (linking to relevant user information). Upon receiving the Bank's message, the ATM stores the active user and changes state to active session.

> ❕ Once a session has begun, the commands below become valid.

Withdraw:

Once a user submits a withdrawal request to the ATM via `stdin`, the ATM sends a message with the request to the Bank. The ATM then changes state to indicate it is waiting for the withdrawal result. The Bank receives this message and checks the active user's balance to see if there is enough to fulfill the withdrawal request. If the balance is not enough, the Bank sends a message indicating that the balance is insufficient. In this case, the Bank does not change state. Upon receiving the Bank's message, the ATM changes state back to active session.

If the balance IS enough, the Bank sends a message instructing the ATM to dispense the relevant amount. The Bank then changes state to indicate it is waiting for a withdrawal confirmation. Upon receiving the message from Bank, the ATM dispenses the money, outputs the successful withdrawal to the active user, and changes state back to active session. It then sends a withdrawal confirmation message back to the Bank. Once the Bank receives this message, it deducts the dispensed amount from the active user's balance and changes state back to open session.

Balance:

Once a user submits a balance request to the ATM via `stdin`, the ATM sends a message with the request to the Bank. The ATM then changes state to indicate it is waiting for the balance result. Upon receiving the ATM's message, the Bank obtains the active user's balance and sends it in a message back to the ATM. No state change is needed for the Bank in this case. Once the ATM receives the balance result, it outputs it to the active user and changes state back to active session.

End-session:

Once the user submits an end session request to the ATM via `stdin`, the ATM sends a message with the request to the Bank. The ATM then changes back to initial state, clearing the active card and user. Once the Bank receives the ATM's message, it also changes back to initial, setting the logged user pointer back to NULL, and clearing the active card.

Mitigated attacks:

Buffer Overflow:

All input from users through `stdin` are sanitized rigorously to be accurate, and all extra/unused input will be discarded. Input is currently limited to 1000 bytes, but it can be adjusted to a smaller size by changing the defined `DATASIZE` macro. As long as `DATASIZE` can fit enough bytes for all valid input, this will not break the functionality of the rest of the code. Refer to the “Sanitizing input” section above for more detail.

Card Forgery:

Since the *.card files are generated by hashing encrypted plaintext, an attacker must know both the key and IV used to produce the ciphertext in order to reproduce the stored hash. However, since the key is unknown to the attacker and the IV is completely unknown, there is no way to reproduce the hash from the plaintext without brute forcing both the key and the IV. Furthermore, our protocol checks this card number when authenticating a user; without the correct card number, an attacker cannot gain access to a user’s account. See “Card Number Generation” for more details.

Message Replay

The ATM and Bank operate on several internal states which limit what kind of messages each can be accept at a given state. For each time a response is expected, the ATM and Bank will change to a state where only a valid response will be accepted. So for any message that the ATM or the Bank is not expecting according to their internal states, that message is ignored. The two programs also use time stamps in each message to determine if a message is created after the last message that the program handled (either sent or received). If an attacker tried to replay a message, that replayed message would have a time stamp that is most likely not valid anymore. Both programs will always store the time stamp of the last message sent or received depending on which took place last.

Message Forgery/Spoofing:

Since messages are encrypted with AES-256 with a secret randomly generated key, an attacker cannot forge valid messages on their own without knowing that key. Also, once a user has been authenticated, each message will always have the same length, so a wiretapper will not know what messages are being sent between ATM and Bank.

Message Splicing:

Since messages are encrypted in CBC mode, they are already resistant to splicing attacks. Also, since each message is encrypted with a different IV randomly generated using OpenSSL's cryptographic random byte generator, the chances of finding two messages with the same IV should be extremely low, rendering it virtually impossible to splice two messages together with some useful effect.

Ignored Attacks:

This design cannot protect against any kind of **denial of service** attacks. The ATM and Bank will hang until the correct message is delivered, so if an attacker decides to mess with the messages so they don't decrypt or become malformed after decryption, the Bank or ATM will just ignore them and continue to hang until the correct message is delivered. If we had more time, we could mitigate this kind of attack by creating a timeout between the two sides and try to recover the session states on both sides. This would allow a continuation of service despite an attack attempt.

Another issue would be our handling of the withdrawal procedure. The Bank requires a confirmation that the funds have been dispensed from the ATM before deducting the amount from the user's balance. We thought this would be more realistic as sometimes there's the possibility that the ATM doesn't have enough specific bills so the user might not get the full requested amount.

This design also cannot protect against **brute forcing someone's PIN**. As PINs are only 4 numerals long, that means there are only 10,000 unique combinations, so given a valid card file, an attacker only needs to check 9,999 combos in the worst case scenario before finding the valid PIN for that user. To prevent this, we could include a max number of times that a user can try logging into an account before it becomes locked and would require a visit to the bank to unlock their account.

General behavior: Bank

For the implementation of the Bank, though attacks cannot exploit improper implementation of the bank

- Bank records the time stamps of every message it sends. Only the time stamp of the last message is kept. In a situation where the Bank does not send a reply to ATM, the time stamps of the last message received from the ATM is recorded instead. The time stamps are set to current time at program start.
- Message time stamps are checked, making sure that the received message was sent at a time after the last received or sent message
- Message account/card number is checked when there's an active session or the Bank needs to validate the PIN. Messages not matching the active user's card number are ignored
- Only when the Bank is at the initial 0 state (NO_SESH) that the card numbers aren't checked because in this state, the only thing that the Bank allows is user login requests.
- Then the Bank processes the rest of the command.
- Bank account specifications:
    - Bank generates the account card number by encrypting the string: "<username>.card<pin>", then hashing it, resulting in 32 hexadecimal characters.
    - If the card number already exists, the bank will continue the above process until a unique number is found.
    - These hex characters are what's in the .card files
    - Card numbers are all unique for each user

General behavior: ATM

- ATM records the time stamps of every message it sends. Only the time stamp of the last message, sent or received, is kept. In a situation where the ATM does not send a reply to Bank, the time stamps of the last message would be the one received from the Bank. The time stamps are set to current time at program start.
- Message time stamps are checked, making sure that the received message was sent at a time after the last received message or sent message

- Message account/card number is checked whenever a general message is received from the Bank. Messages not matching the active user's card number are ignored
- For initial user verification messages, there is no active card number at the time of processing, so card number is not checked upon receipt from the Bank
- Then the ATM processes the rest of the command
- If a message is rejected, the ATM waits until a valid message is received before continuing execution

Protocol Flow in Detail (Bank and ATM)

> ❕ Arrows indicate sender/receiver of each message

Initially, both Bank and ATM start at state 0 (INITIAL and NO_SESH respectively).

atm→bank: ATM begins session with a username, sends it to Bank

- sends *initial* command: **<username>**
- ATM changes state → 11 (VERIFY_USER_WAITING), waiting for confirmation of user
- Bank checks for user existence in records

bank→atm: Bank replies to user login request after searching for user

- (IF USER NOT FOUND) sends *initial* command: **no-user-found**
  - Bank does not change state
  - ATM changes state → 0 (INITIAL)
- (IF USER FOUND) sends *initial* command: **user-found**
  - Bank changes state → 99 (AWAIT_PIN)
  - Bank stores username into bank→active_user
  - ATM stores card number in atm→active_card: a char array of 32 hexadecimal characters

- ATM receives message from bank, reads PIN from stdin

  atm→bank: ATM sends result of PIN read to Bank

  - (IF PIN INCORRECT FORMAT) sends command: **unverifiable**
    - ATM clears card number out of atm→active_card
    - ATM change state back → 0 (INITIAL)
    - Bank receives message and clears bank→active_user
    - Bank changes state back to → 0 (NO_SESH)
  - (ELSE) sends command: **verify XXXX** (PIN to verify, 4 chars)
    - ATM changes state → 22 (VERIFY_PIN_WAITING)
    - Bank receives message, stores card number in bank→active_card: a char array of 32 hexadecimal characters
    - Bank receives PIN in message, checks stored PIN

      bank→atm: Bank sends result of PIN comparison to ATM

      - (IF NOT EQUAL) sends command: **access-denied**
        - Bank clears card number out of bank→active_card
        - Bank changes state back to 0 (NO_SESH)
        - ATM receives message
        - ATM clears card number out of atm→active_card
        - ATM change state back to 0 (INITIAL)
      - (IF EQUAL) sends command: **access-granted**
        - Bank changes state → 11 (OPEN_SESH)
        - Bank saves a pointer to the logged user's file (bank→logged_user)
        - ATM receives message
        - ATM stores the user in atm→curr_user
        - ATM changes state → 33 (ACTIVE_SESSION)

❗ ALL MESSAGES BELOW ARE ONLY APPLICABLE IN **access-granted** CASE I.E. WHEN THERE IS AN ACTIVE/OPEN SESSION

atm→bank: ATM sends user's request to withdraw money

- sends command: **withdraw\0(<int> four bytes)**
  - ATM changes state → 44 (WITHDRAW_WAITING)

    bank→atm: Bank sends result of withdraw attempt back to ATM
    - (IF BALANCE NOT ENOUGH) sends command: **insufficient**
      - Bank does not change state
      - ATM receives message
      - ATM changes state back → 33 (ACTIVE_SESSION)
    - (IF BALANCE ENOUGH) sends command: **dispense\0(<int> four bytes)**
      - Bank changes state to → 33 (WITHDRAW)
      - ATM receives message and dispenses money
      - ATM changes state back → 33 (ACTIVE_SESSION)

      atm→bank: ATM sends confirmation of money dispensed
      - sends command: **dispensed\0(<int> four bytes)**
        - Bank deducts the dispensed amount from the active user's balance
        - Bank changes state back to 11 (OPEN_SESH)

atm→bank: ATM sends user's request to check balance to Bank

- sends command: **balance**
  - ATM changes state → 55 (BALANCE_WAITING)

- Bank receives message and obtains balance

bank→atm: Bank sends balance back to ATM

- Bank sends command: **balance\0(<int> four bytes)**
- Bank does not change state
- ATM receives message
- ATM changes state back → 33 (ACTIVE_SESSION)

atm→bank: ATM sends user's request to end session to Bank

- sends command: **end-session**
  - ATM changes state → 0 (INITIAL), clears atm→active_card and atm→curr_user (username of logged in user)
  - Bank receives message
  - Bank changes state → 0 (NO_SESH), sets bank→logged_user (pointer to the user's file) to NULL, and clears bank→active_card