Team 3
Ka Wong

Built-It Design Document

1. Protocol Overview

Our protocol begins in bank-main.c, where we create a bank. Upon creation, the bank creates and saves an auth file, which is used to authenticate connections between a valid bank and atm. In our case, the auth file is a symmetric key, consisting of 256 randomly generated bytes which we assume is shared safely between all valid atm and bank instances.

Once we create a bank, the bank begins waiting for incoming connections which will be processed one at a time as they are received. The steps for each connection are the following and are assumed to occur in order:

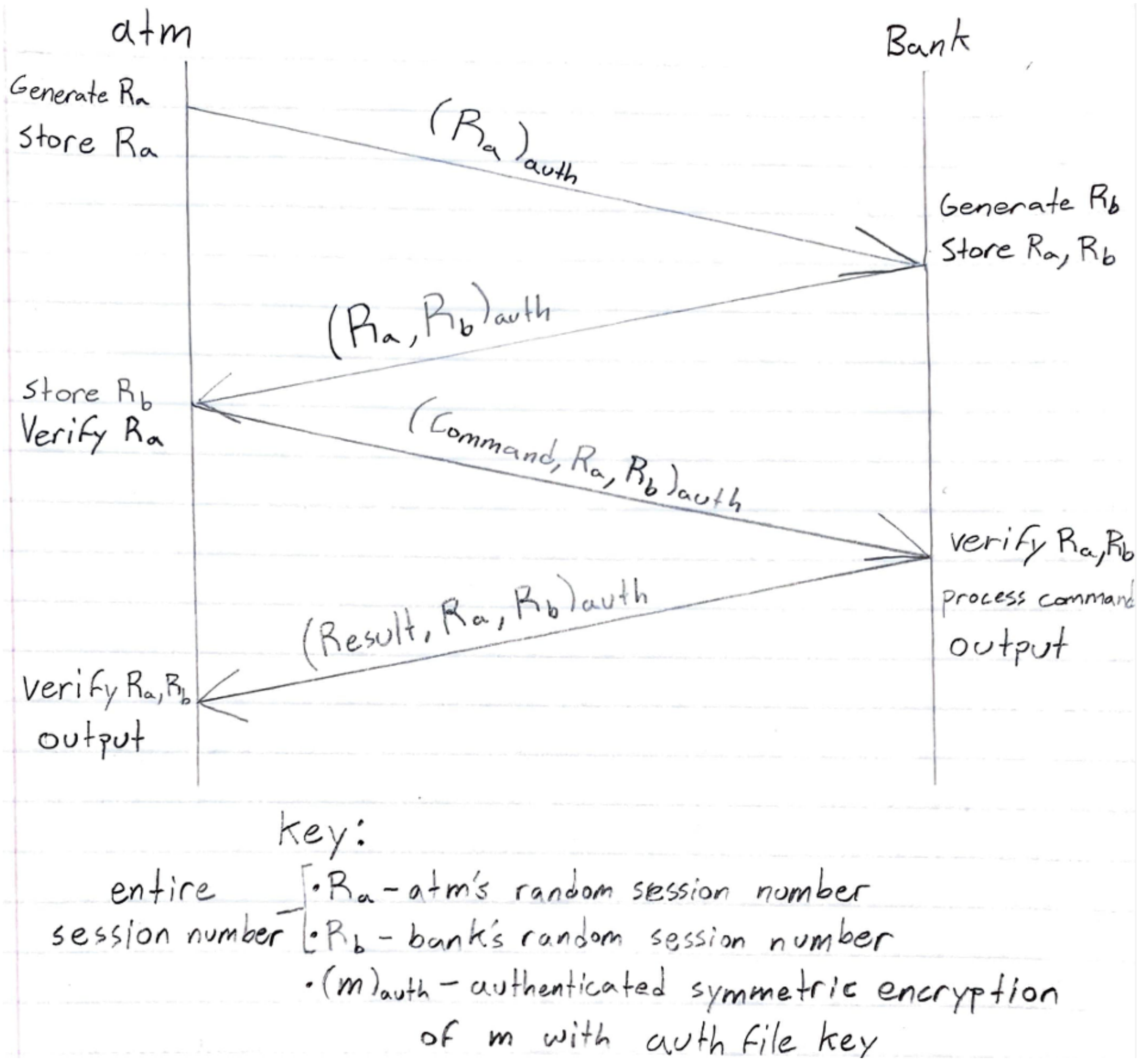
0. Before initializing any connection, atm validates all command line inputs to meet specifications and POSIX standards (also done in bank to validate bank options upon creation). Input validation contains both regex checks of format and bounds checks for numeric inputs. Also, if all inputs are validated, and the mode of operation is to create a new account, a card is generated. The card is saved in the appropriate location only once the communication with bank is completed successfully. A card is 512 randomly generated bytes and is associated with a specific account. Also, when a bank is created, it randomly generates a 256-byte symmetric key that is placed into an auth file that we assume is shared securely for every bank and atm. This, along with an authenticated encryption scheme, ensures that messages are confidential, cannot be modified without either party knowing, and are authentic (only bank and valid atms have the auth file)
1. The atm encrypts a randomly generated session number of 16 bytes with its copy of the auth file. The atm uses authenticated encryption in GCM mode with a randomly generated IV. The atm then sends the tag, IV, and encrypted session number to the bank.

Session numbers are used to ensure that past communications between bank and atm are invalid since they change each connection.

2. Once the bank retrieves the message from atm, it decrypts the message using the shared auth file symmetric key, the received tag, and IV. If the message fails to decrypt, then bank terminates the connection and continues to wait for new communications. Otherwise, it generates its own session number of 16 bytes and appends that to the atm's provided session number. It then encrypts the combined session numbers (32 bytes) using the same encryption scheme above and sends the new tag, new IV, and encrypted combined session number.
3. Once atm receives this message, it decrypts this using the auth file. If it fails to decrypt, atm sends an empty failure packet to bank and returns 255. This packet ensures bank will terminate the connection and continue, rather than timing out and terminating bank. This failure packet is sent at any point in atm where there is a failure that will cause atm to exit while bank is still awaiting communication. Otherwise, atm checks that the first 16 bytes of the message (its original session number), matches. If it does, atm has authenticated this as a new connection with the bank. The atm then creates a packet containing the information from the command line prepended with the full 32-byte session number. The atm then encrypts this with the same scheme above and sends it to the bank using a new tag and IV.
    a. The sent packet has the following structure: IV (12 bytes) || tag (16 bytes) || encrypted(combined session number (32 bytes) || mode of operation (4 bytes) || account (123 bytes) || content of card file (512 bytes) || amount(8 bytes stored as a whole and decimal part))

4. Once received in the bank, the bank decrypts the message and checks that the session numbers match. If this fails, then bank terminates the connection and continues to wait for new connections to be established. Otherwise, the bank has successfully authenticated the atm. It then breaks apart the packet sent by atm and processes the command. Once the command is processed, the bank prepares its response, which is either “ERROR” (if the command failed, like withdrawing too much) or the string to be printed at atm. The response is prepended with the full session number, encrypted using the auth file, and sent to atm with the new tag and IV. Importantly, all responses sent by the bank here are the same size (“ERROR” or not). Bank concludes and proceeds to its next connection.
    a. The bank is able to preserve the state of accounts using a hashtable stored in memory. The hashtable stores account structs, which have an account name, card file content, balance, and a pointer to the next account (used for bucketing accounts that hash to the same index). The hashtable has a size of 256 such that the index is a one-byte value generated from hashing the account name using sha256. Also, accounts are searched for according to their name and then verified using card file content. A hashtable was chosen to efficiently search for accounts.
5. The atm then decrypts the received message using the supplied IV and tag, then checks that the session numbers match. If they do not, atm fails and returns 255. Otherwise, the atm interprets the result. If the result is “ERROR”, the atm fails and returns 255. If the result is anything else, the atm prints the result of the executed command and exits successfully. At this point, if the mode of operation is to create a new account, atm also writes the card content it generated earlier into the card file.

All random value sizes were chosen to be large enough that they cannot be reasonably guessed/brute-forced and with randomness to ensure they cannot be predicted (rather than other options, like counting). The size of other fields was determined by the largest possible value it could contain. Other design choices were either listed in the design specifications or were chosen to protect against possible attacks. The reason for these choices is further justified in the sections below.


atm
Bank
Generate R_a
Store R_a
$(R_a)_{auth}$
Generate R_b
Store R_a, R_b
$(R_a, R_b)_{auth}$
Store R_b
Verify R_a
$(\text{Command}, R_a, R_b)_{auth}$
verify R_a, R_b
process command
output
$(\text{Result}, R_a, R_b)_{auth}$
verify R_a, R_b
output
key:
entire session number
• R_a - atm's random session number
• R_b - bank's random session number
• $(m)_{auth}$ - authenticated symmetric encryption of m with auth file key

Waterfall Diagram of packet protocol

2. **Attacks**

   a. **Invalid Command Line Arguments**

      i. One potential attack would be for a malicious actor to execute the atm program with disallowed inputs to cause undesired behavior. For instance, a malicious user with a valid card file and account name may attempt to withdraw negative money in order to increase their account balance. Also, an attacker may try to underflow a balance by withdrawing, so that it becomes the maximum value instead.

      ii. To address this, our protocol has the atm validate all incoming input on the command line. This includes regex and bounds checking to ensure commands sent by the atm to the bank follow only the allowed format, including allowing only positive amounts. Inputs formatted validly that should still be disallowed on the bank side (such as accessing a non-existing account or withdrawing more than you have) are stopped there and an error is sent to atm.

   b. **Impersonating Users**

      i. Another potential attack is for a malicious actor to attempt to impersonate a valid user, such as withdrawing from an account that is not theirs. A potential method may be to brute force or guess account names and/or card files.

      ii. While we have no control over what account name a user chooses, we mitigate the impersonation of users by randomly generating a 512-byte card file for each account. In order to validate a command, the given card file must exactly match the one associated with the account in bank. Unless the attacker has access to the card file itself, it is highly unlikely that they would be able to guess or even brute force a 512-byte card file for a given account (~2^(512 * 8) possibilities).

c. Buffer Overflow

- **i.** A potential attack would be to attempt to overflow buffers by sending an excessive amount of bytes over the network or even by specially crafting command-line arguments to do so. Buffer overflows have the potential to overwrite or reveal arbitrary memory which could expose details like the auth file.
- ii. To prevent this, whenever we receive data at either end, we ensure that we read a specific number of bytes into a buffer of that exact size. That means that regardless of what an attacker may send, we only read in the amount we want to. This is further mitigated as decryption will fail when receiving packets of invalid size. Also, whenever we treat the data as a string, we ensure that it has been sent by an authenticated user who has validated the information and the data has not been modified since.

d. Stealing Account Information With Man in the Middle (MITM)

- i. Another potential attack would be a MITM attack where an attacker observes packets being sent between atm and the bank. The attacker would be able to see exactly what commands the atm was sending, including the user's account name and card file content. They could then use this content to impersonate the user.

   ii. We solve this by encrypting all data sent from both the atm and bank. The attacker cannot retrieve the data unless they have access to the key, which we store in the auth file (that we assume is securely located at both atm and bank). To ensure that encryption is not predictable, we generate a new random IV every time a packet is sent and we use GCM mode.

e. **Getting Information From Packet Size**
   i. Another potential attack would be to analyze the packet size and use that to determine what commands an atm might be sending or what results bank is returning. For instance, a get command could potentially always send fewer bytes, since it does not require an amount. This could disrupt confidentiality.
   ii. To counter this, we use a uniform packet size. That is, regardless of the command or information sent, the packet is always the same size as specified in the protocol. The attacker can determine what type of packet it is by the order it is sent in, but would not have an idea of what it contains (assuming it is encrypted).

f. **MITM Integrity Attacks**
   i. Another potential attack is if the attacker modifies the packet as it is being sent to cause undesired behavior. For instance, modifying the bytes so that a user withdraws more than they expected or changing the command type.
   ii. To protect against this, we use authenticated encryption and send the tag. This ensures that if an attacker modifies the bytes of a sent message, then decryption will fail, preventing modified inputs from proceeding.

g. **Replication Integrity Attacks & Impersonating ATM and Bank**

i. Another potential attack would be an attacker attempting to impersonate communications from the bank or atm. This can be done through replication, as an attacker could copy a past packet and send it to either atm or bank to imitate a valid command. For instance, if a packet told the bank to withdraw a certain amount, the attacker could copy that packet and send it to the bank to withdraw that amount again, as the packet was already formatted and encrypted.

ii. Firstly, authentication is handled with the auth file, which contains a symmetric key shared between valid atms and banks. This key allows us to defend against impersonation since only a valid atm/bank with access to the key would be able to encrypt or decrypt the communications between them. To defend against replication, both bank and atm generate a random session number. In order to create an authenticated connection, the bank must prove that it knows and decrypts atm's session number, and atm must do the same for the bank's session number. Also, a session number is only valid for a single session, so if a command is reused from another session via replication, it will fail as the session numbers are random. Also, if an adversary sends a packet in the middle of a session, that packet will be denied by both ends due to either decryption errors or a mismatching session number. This will disrupt the session, damaging availability, but integrity and confidentiality will be preserved as the connection terminates before any malicious modifications take place.

h. **MITM Availability**

   i. If an attacker was able to modify a packet when it is being sent to the atm, atm will then fail to decrypt and will exit. Bank could then timeout if it is still waiting for another packet from atm.

   ii. To solve this, if atm fails when the bank is still waiting for a packet, atm will send an empty packet that will then cause bank to terminate the connection and return to waiting for new connections (instead of crashing from timeout). This mitigates the availability attack, but other types are still possible as described below.

3. **Unmitigated Attacks**

   a. **Availability**

      i. Our current approach does not mitigate against a variety of availability attacks. These include DDoS, deleting packets, and attacking availability via replication. Regarding DDoS, we have no protection against attackers sending many requests, as our bank will continue to serve them one by one, thus denying service to valid users. It is also possible that an attacker could take down packets as they are sent from atm to bank or vice versa. Since the atm/bank cannot proceed until they receive the message, this would halt service. Finally, it's possible for an attacker to replicate only the first communication, which does not require any authentication yet. The bank would then serve the request and await a response, which an attacker may never send.

ii. These attacks are unmitigated because the networking constraints are not defined by our protocol. For instance, we cannot adjust bandwidth, or we cannot block certain IPs. However, we can cause the bank or atm to timeout if they do not receive a communication in time, which would allow the atm or bank to exit if the attacker halts communication. We did take this approach, however, this does not solve the core availability issue. An attacker could continue to timeout the bank/atm, denying service to valid users.

b. Hashing Passwords in Bank

i. If an attacker gets access to bank's memory, then all card content and account information is available in plain.

ii. A potential solution would be to store a salted multi hash of each account's card content in the long-term storage of bank. For each connection, the bank could then compute the salted multi hash and verify that instead. This mitigates such attacks since plain card content would only be available during the duration of a connection with atm. We did not protect against this attack since it assumes a high privilege attacker that our threat model did not cover. Also, if the attacker had access to memory they would just be able to use the auth file stored in bank anyways. In our current implementation, we also store auth file in memory for the duration of bank. To minimize the time auth file is in memory we could read the file for each communication. However, an attacker can force the auth file to be in memory by simply sending a packet to bank. Thus this change would have minimal benefit.