ATM System Protocols and Design

Team: Christine Schroeder and Simon Milburn

1. Protocol

1.1 Init

command line usage: bin/init <filename>

Initialization file creates two different files, each with the same private key for encryption of packets. These initialization files are each passed in as an argument to their prospective program; the bank takes the .bank file and atm takes the .atm file. Init also takes in one command line argument that will serve as the actual name of the files for example <name>.bank and <name>.atm, along with the path to that file.
Print upon success: Successfully initialized bank state

.bank and .atm files

These two files contain the shared secret that the bank and atm will use to encrypt packets and communicate. This secret key is a randomly generated 32 character string. All characters are lower-case letters.

1.2 Bank

command line usage: bin/bank <filename>.bank

The bank program is used to create new user accounts as well as edit an account. Unlike the ATM, the bank program does not require the participant to authorize by entering a PIN upon entering since we assume that an adversary would only indirectly attack the bank through the ATM.

The program first creates a bank object which sets up all communications in order for the bank to listen for connections from the router. Two hash tables are defined within the main function, one named *balance* and the other named *users*. *Balance* hash table stores the current balances for all users and is manipulated when changes to the balance are made. *Users* hash table stores keys for each user's card file.

User keys are generated in the same format as the key in the init program (random 32 character string); however, each user's key is unique. It is important to note that the PIN is not stored at the bank.

The bank continuously listens for connections. If a socket is created, then the bank knows that the message they are receiving is remote and the message is encrypted. This conditional decrypts the message using the private key from the .bank file and sends it to a function that deals with remote calls. Otherwise, it is a local command and is dealt with differently.

The encryption that will be used is AES 256 CBC with the appropriate key and an IV of 32 zeros. Both the remote call function and the local call function pass in the two hash tables, the bank object, and the sizes of the messages as parameters. The remote call function also passes in the key from the .bank file in order to encrypt messages to send back to the ATM. All remote calls must adhere to a strict format in order to be considered valid. If they are not valid, a NULL packet is sent back to the ATM in the correct format. All interactions with the ATM will be discussed in more detail in section 1.3.

create-user

command line usage: create-user <user-name> <PIN> <balance>

This local command creates a user at the bank. After checking for malformed input the username and initial balance are stored in one of the two appropriate hash tables. A random 32 character string is generated and will serve as the unique key for that user's account. It is used to encrypt and decrypt the user's card file that is created. The card file contains a string in the format <user-name>;<PIN> and is saved as <user-name>.card. The card is encrypted with the user's key so no one other than the user can decrypt the card.
Print upon success: Created user <user-name>

deposit

command line usage: deposit <user-name> <amount>

This local command deposits an amount in <user-name>'s account by accessing the balances hash table. In order to prevent overflow from input greater than UINT_MAX, amounts are initially stored as a long long the cast back to an unsigned int. Based on the functionality of the hash table, the entry containing the old balance must be deleted and the user and new balance must be added. After checking that the deposit will not overflow the balance, the new balance is updated in the hash table.
Print upon success: $<amount> added to <user-name>'s account

balance

command line usage: balance <user-name>

This local command accesses the balance hash table to print out <user-name>'s current balance. The hash_table_find() function is used to look up the requested user's balance.
Print upon success: <balance>

1.3 ATM

command line usage: bin/atm <filename>.atm

The ATM program is used for someone who wishes to open their already created account remotely (not at the bank). The program first creates an ATM object. The ATM itself

stores no account information about each user so it is completely dependant on communication with the bank program for this. The communication, in this case, is in the form of encrypted packets that are send back and forth from ATM to bank. The ATM file created in init is used for encryption.

The encryption that will be used is AES 256 CBC with the appropriate key and an IV of 32 zeros. Each user must have a valid card and a four-digit PIN number in order to open their account. The ATM has a session_token which is used to know which user is logged in (if there is one) and who's account to modify at the bank. This session_token is a static variable, so it exists beyond the lifetime of the process command functionalities.

Many of the ATM's commands require account information about a specific user, so the ATM must send a request to the bank for information. These requests are strictly formatted packages. All packets begin and end with <> and each piece of information within the packet is delimited by a |. Typically packets hold three pieces of information; the first piece is the request type, the second piece is the user-name, and the third piece is the usually an amount being withdrawn. If the ATM receives a packet that is invalid, it will print something like "Invalid Packet" and discard the packet. The following part will cover each specific packet being sent for each command.

begin-session

command line usage: begin-session <name>

This command is used to authenticate the user that is trying to log in. First it encrypts and sends a packet to the bank in the format <authentication|name>. The bank receives and decrypts the packet. The bank then searches for the name in the user key hash table. The bank encrypts and sends a response packet to the ATM in the format <authentication|user_key>. If the user does not exist, the bank encrypts and sends <authentication|not found>.

There is a hash table, called *tries*, that keeps track of the amount of attempts at a PIN that a user makes and if it exceeds the maximum, the account is locked out. Currently, the maximum number of attempts is set at three. This prevents brute force attacks at all the possible PINs. If the correct PIN is entered within the three tries, the hash table entry is reset to 0.

The ATM receives and decrypts the bank response packet. If the user key is in the packet, it then searches for the user's card file. If the card file is found, the user key sent back from the bank is used to decrypt the card. The user is prompted to enter their PIN at this point. Once the card file is decrypted and parsed, the PIN in the card file and the PIN entered are compared. If they match, then the user name is set as the session_token; the session token is also returned to main and is used to signify that that the user is logged in. As stated, the session_token is saved even after termination of the process command function, but the information sent back from the bank pertaining to a certain user is not stored.
Print upon success: Authorized

withdraw

command line usage: withdraw <amount>

This command modifies the already logged in user's balance by taking out amount. The ATM encrypts and sends the bank a packet in the form <withdraw|user|amount>. The bank receives and decrypts the packet. The bank then gets the current balance of the user specified in the packet from the balances hash table. The amount is subtracted from the current balance yielding a new balance. Following the hash table protocol, the old user entry must be deleted from the balances hash table and a new entry with the updated balance must be added. The bank encrypts and sends a response packet to the ATM in the format <withdraw_successful>. If there is a problem and the user does not have enough money to take out amount, then the bank encrypts and sends a response packet to the ATM in the format <Insufficient funds>. The ATM receives and decrypts the packet.
Print upon success: $<amount> dispensed

balance

command line usage: balance

This command prints out the balance of the current user. The ATM encrypts and sends the bank a packet in the form <balance|user>. The bank receives and decrypts the packet. The bank then gets the current balance of the user specified in the packet from the balances hash table. The bank encrypts and sends a response packet to the ATM in the format <balance|user|amount>. The ATM receives and decrypts the packet.
Print upon success: <$balance>

end-session

command line usage: end-session

This command ends the session with the current user. The session token is set to empty and the function returns a blank session token. This command requires no information from the bank, so no packets are sent.
Print upon success: User logged out

2. Attacks and Security Measures

1. Attack: overflow the commands
   Defense: each command in the bank is saved in a buffer that is the maximum line length then the value length is checked and copied into a variable using strncpy.
2. Attack: sniffing packets between bank and atm
   Defense: All packets are sent using aes cbc 256 with private key crypto. The private shared key between the bank and the ATM is established in the init files. Each packet has a strict specific format. It starts with '<' and ends with '>' and each piece of info in the packet is delimited by '|.' This means that if a packet is received and any of those characters are not present or the packet format is not correct, then the packet will be

considered “tampered with” and will be discarded. Similarly, if the bank gets a remote call that it can’t decrypt because it has been with, it should send a null packet back to atm since the atm is expecting some sort of packet back no matter what.

3. Attack: Attacker attains card and doesn’t know pin
   Defense: The pin is stored on the card so once the card is decrypted, it checks to see if the pin on the card matches the pin entered and it will not be. There is also a max number of attempts to prevent brute forcing the PIN.
4. Attack: attacker tries to edit card file to change user pin
   Defense: the card is encrypted using aes cbc 256 and with a specific user key that is stored at the bank. If the encrypted card is edited, the key will not be able to decrypt the card and the user will not be authorized.
5. Attack: attacker knows that bob has an account at the bank and has his pin but they don’t have their card
   Defense: The ATM checks to make sure that the card file exists AKA if the participant has the card and if it doesn’t then no access is granted. Also, the card is the only place where the pin is stored so if the participant does not have the card, then there’s no way to verify the pin.
6. Attack: attacker tries to inject a packet to deposit money into the user’s account
   Defense: Without knowing the shared secret between the bank and the atm that is used to encrypt the packets, this would be impossible. The packet would be received and would not be able to be decrypted; then it will be discarded.
7. Attack: Withdraw more money from a user’s account than the balance permits
   Defense: check if the balance-withdraw_amount is less than zero which would make it invalid.

Notes:

We chose to ignore the idea of banning connections from a malicious sender if messages sent cannot be decrypted. In this scenario, this would be unimportant because the bank only supports packets from one ATM.

We did not include an unlock functionality once the user on the ATM is locked out. The bank is constantly listening on the network from the ATM but the ATM is not always listening for the bank. Therefore, an unlock command could not be made without an initial signal from the ATM.