Design Doc

Init

In `init`, we use OpenSSL to generate a 4096-bit RSA private key that we store in our `.bank` and `.atm` files. We use this shared private key to encrypt and decrypt information between the bank and ATM to prevent man-in-the-middle attacks from decrypting intercepted information.

Bank

- The Bank struct is composed of networking states, the RSA key for encryption, a nonce, hash tables to maintain PINs, balances, card numbers, and a list of logged in users.
- `bank_create()` initializes a Bank struct and returns a pointer to the Bank. The function sets up the networking state, sets the nonce counter to 0, and creates hash tables for PINs, balances, card numbers and a list of logged in users.
- `bank_free()` cleans everything up, freeing all memory allocated for the bank instantiation.
- `bank_send()` and `bank_recv()` encrypt and decrypt outgoing and incoming data using the private key stored in our `.bank` file.
- Helper functions `process_create_user_command()`, `process_deposit_command()`, and `process_balance_command()` are utilized by `bank_process_local_command()` to support the `create-user`, `deposit()`, and `balance` commands that can be run directly on the bank server.
    - `process_create_user_command()` -
        - Parameters: bank processing the command, create user command string
        - Functionality: creates a user if it does not already exist in the specified bank. Does so by parsing the command input for the username, pin, and balance and checking if the username exists in the bank. Prints out a message upon error.
    - `process_balance_command()` -
        - Parameters: bank processing the command, balance command string
        - Functionality: prints out the specified user's current balance. Does so by parsing the command input for a username and searching for the user in the bank. Prints a message upon error.
    - `process_deposit_command()` -
        - Parameters: bank processing the command, deposit command string
        - Functionality: deposits the specified amount into the user's account. Parses the command string for a username and amount then adds the amount to the user's account. Prints a message upon error.

- `bank_process_remote_command()` supports the `CheckSession, BeginSession, Withdraw, Balance`, and `EndSession` commands that are sent from the ATM. These commands from the ATM are handled as follows:
    - `CheckSession` - verifies that a session can be started by checking to see if the user exists and that a user is not already logged in. If any of these cases are true, then a message will be sent back to the ATM indicating what the issue is.
    - `BeginSession` - once the session is checked using `CheckSession`, the program only then calls the `BeginSession` command. The user's credentials are verified and the function then returns information to the ATM on whether or not the user is authorized.
    - `Withdraw` - revalidates the user's authorization and checks that the user is logged in. Attempts to withdraw the specified amount and indicates success if withdrawn or failure if there were insufficient funds. Relays to the ATM the outcome of the command.
    - `Balance` - revalidates the user's authorization and checks that the user is logged in. Sends the user's current balance back to the ATM.
    - `EndSession` - revalidates the user's authorization and checks that the user is logged in. Once the user has completed their session, the current session is terminated and the user is logged out. Relays to the ATM the outcome of the command.

ATM

- The ATM struct is composed of networking states, the RSA key for encryption, a nonce, a username, card, and PIN.
- `atm_create()` initializes an ATM struct and returns a pointer to the ATM. The function sets up the networking state, and sets the username to NULL and the nonce counter, card, and PIN to 0.
- `atm_free()` cleans everything up, freeing all memory allocated for the ATM instantiation.
- `atm_send()` and `atm_recv()` encrypt and decrypt outgoing and incoming data using the private key stored in our `.atm` file. Note that the nonce counter is incremented in `atm_send()`.
- Helper functions `process_begin_session_command()`, `process_withdraw_command()`, `process_balance_command()`, `process_end_session_command()` are utilized by `atm_process_command()` to support the begin-session, withdraw, balance, and end-session commands. Each of these functions send data packets to the bank encapsulating the command information which is

then received and processed by the `bank_process_remote_command()` function outlined above.

- `process_begin_session_command()` -
  - Parameters: current atm state, number of arguments, argument vector
  - Functionality: attempts to log a user into a new session. This is done in two steps, calling the `CheckSession` command to verify the existence of the user and that they are not already logged in. Then the `BeginSession` command is sent to the bank to actually authenticate the user.
- `process_withdraw_command()` -
  - Parameters: current atm state, number of arguments, argument vector
  - Functionality: withdraws a certain amount from a user's account. Sends the `Withdraw` command along with the amount and account information to the bank which then processes the withdrawal.
- `process_balance_command()` -
  - Parameters: current atm state, number of arguments, argument vector
  - Functionality: gets the current balance of a user's account. Sends the `Balance` command along with the user information to the bank which then returns the user's balance.
- `process_end_session_command()` -
  - Parameters: current atm state, number of arguments, argument vector
  - Functionality: logs the user out of their session. Sends the `EndSession` command along with the user information to the bank which then attempts to log the user out.

Attacks Considered

1: Man-in-the-Middle Attack (Implemented)

- Vulnerability
  - A Man-in-the-Middle Attack involves intercepting data in transit, reading that data, and in some cases modifying the data before it reaches its intended destination. In the case of our Bank and ATM, the ATM and Bank send packets to each other through a router. If these packets are unencrypted, attackers can intercept, read, and modify messages between the Bank and ATM. This poses the risk of attackers gaining user PINs or other user information and potentially sending additional unauthorized commands that could modify the user's balance.
- Our Solution
  - Our solution to a Man-in-the-Middle Attack was to add encryption and decryption. With our init file, we generate a 4096-bit RSA private key that is

shared between both the Bank and ATM to encrypt and decrypt information passed through the router. So, whenever the Bank sends information to the ATM or vice versa, we encrypt the information before sending. Then, whenever receiving information, we decrypt it with the same key. Encryption is seen being performed in bank_send() and atm_send(), and decryption is performed in bank_recv() and atm_recv(). Because we encrypt messages before sending them, attackers that intercept the messages will intercept encrypted information. Because RSA-4096 is secure according to the NIST (for now), attackers will not be able to decrypt the information to read sensitive user data without knowing the key.

- Our original RSA implementation would produce the same ciphertext given the same plaintext and same key. This method of encryption was vulnerable to chosen-plaintext attacks where attackers send specific messages to reveal relationships between the plaintext and ciphertext and potentially learn sensitive information or send replay attacks similar to the Cryptography Project. As such, we added PKCS#1 OAEP padding to our existing RSA encryption code, which pads the plaintext with random bytes and protects us against known ciphertext attacks and replay attacks.

2: Buffer Overflow Attack (Implemented)

- Vulnerability
  - When reading in user input for fields like username, PIN, and balance, a buffer overflow involves input exceeding our allocated space for the field. For example, the username is restricted to 250 characters, a buffer overflow attack could involve sending more than 250 characters and possibly overwriting the stack's return address like in project 1.
- Our Solution
  - When reading in user input, we limit the size of the input. In process_create_user_command(), we have sscanf(command, "create-user %250s %4d %d", username, PIN, balance). The %250s prevents reading in more than 250 characters for the username and %4d prevents reading in more than 4 digits for the PIN. With the balance, if a user inputs a balance that exceeds the max size for an int, the sscanf() function would read in an invalid number that is filtered out when checking balance < 0. Overall, we do similar restraints when reading in user input with sscanf() in process_deposit_command(), process_balance_command(), process_begin_session_command(), and process_withdraw_command(). Since we limit the size of user inputted data, attackers are NOT able to overflow the allocated space and overwrite information like return addresses.

3: Replay Attacks / Fraudulent Commands (Implemented)

- Vulnerability
    - A replay attack involves an attacker intercepting a message and resending it. For example, an attacker could intercept a deposit message and replay the deposit message to seem like they deposited $100 twice rather than once.
- Our Solution
    - Our solution to identify fraudulent commands was to use a nonce counter. In cryptography, a nonce is a random or pseudo-random number that authentication protocols use in communications to protect private communications by preventing replay attacks. A nonce introduces randomness into communications from the ATM so that the Bank can verify the user. This added uniqueness prevents attackers from using prior communications to impersonate legitimate Bank account holders. If requests to the Bank contain the wrong nonce, they are rejected. Our nonce counter prevents replay attacks that rely on impersonating prior communications in order to gain access.
    - A limitation to our nonce implementation is that we use a counter, starting at 0 and incrementing by 1, which could be predictable to attackers and allow them to send reply attacks by "guessing" the nonce. A solution to nonce predictability that we did not implement involves randomizing the nonce rather than using a counter.

4: Invalid Command Input (Implemented)

- Vulnerability
    - An attacker could purposefully provide invalid inputs to executed commands in order to produce unexpected behavior with our Bank or ATM. If our Bank and ATM programs do not properly validate command arguments before attempting to change account balances or complete any procedures, this could cause unexpected behavior or corrupted data which could affect subsequently executed commands.
- Our Solution
    - When reading in user input, we utilize the sscanf function to validate all arguments before proceeding. In process_create_user_command(), we have sscanf(command, "create-user %250s %4d %d", username, PIN, balance). The %250s requires the first argument to be no more than 250 characters for the username, %4d requires the second argument to be a 4-digit PIN, and %d requires the third and last argument to be a number (series of digits) representing the account balance. With the balance, if a user inputs a balance that exceeds the max size for an int, the sscanf() function would read in an invalid number that is filtered out when checking balance < 0. Overall, we do similar restraints when reading in user input with sscanf() in process_deposit_command(), process_balance_command(), process_begin_session_command(), and

process_withdraw_command(). Since we validate all user inputted data, attackers are NOT able to trigger unexpected behavior through malformatted command arguments.

5. Format String Attacks (Implemented)

- Vulnerability
    - An attacker might attempt to input format specifiers like %p to reveal information about the stack or program. For example, the attacker could create a username with the format specifier %p. Then, if there was a line like printf(username), then the attacker could find the location of the username on the stack.
- Our Solution
    - To prevent format string attacks, we avoided user controlled format strings. After creating a user in process_create_user_command(), we print the username with printf("Created user %s\n", username). By using the format specifier %s, we print the string stored in the username without interpreting the username as a format specifier. So, if an attacker were to create a username “%p”, then we would simply treat their name as a string rather than a format specifier. In short, we avoid user controlled format strings throughout our program by using a format specifier like %s when printing to prevent fields like username from being interpreted as a format specifier.

6: Brute Force Login Attack (Not implemented)

- Vulnerability
    - An attacker might try a bunch of combinations of credit cards, PINs, and usernames.To find a valid username, attackers could try to create a new account with usernames until they receive an error message that the user already exists. For the credit card, the attacker could try a brute force method for this or might attack a user’s stolen card. Then, the attacker could brute force and guess the PIN of the ATM since the 4-digit PIN is limited to 10000 combinations. If the attacker is successful, the attacker is able to perform actions on behalf of the victim, potentially withdrawing their entire balance.
- Potential Solutions
    - To help prevent brute-force attacks, we could limit the number of attempts to log in to 5 or some low number so that the attacker will not be able to attempt so many combinations. An additional measure to protect users would be preventing easily guessable information. For example, we blacklist easily guessable usernames like “username” and PINs like 0000 or 1234.