:0 (Group 6) ATM Design Document

Andrew Liu, Bao Ngo, Keva Singhal, Victor Lee, Zisheng Chen

Goal

Build and establish a system that has external ATMs which facilitate secure money deposits and withdrawals. When each user registers with the bank and creates their account, they receive a card with a distinct username and pin to verify their identity with the bank. This identification is used to validate their identity when going to an atm. Once a user is able to login through the ATM, they can withdraw money and check their balance. In any system where programs communicate with each other, security measures are added to prevent unauthorized access and attempts to steal personal information.

Design Overview

I INIT

The initialization program sets up two files that set-up the state of BANK and ATM. Init runs when the user calls `<path1>/init <path2>/<init-fname>`, where `<path1>` and `<path2>` are user-defined directory paths and `<init-fname>` is an assigned name to the files. Rather than hard-coding paths, the user can specify where these files live. For example, if a user inputs `~/gitsrc/atm/init /tmp/foo`, then init should create `/tmp/foo.bank` and `/tmp/foo.atm`. One thing to note, if `/tmp` or any other user-specified directory does not exist, then the program will create the directory.

If a user fails to provide only one argument, then it will print `Usage:  init <filename>` and return value 62. If either file `<path2>/<init-fname>.atm` or `<path2>/<init-fname>.bank` exist, then it will print `Error: one of the files already exists` and return value 63. There are no side-effects if this error is returned. Any other errors initializing the file should result in the program printing `Error creating initialization files` and return value 64. Once the bank is successfully created, it will print `Successfully initialized bank state` and return value 0. The implementation accounts for each of these cases and should successfully follow the protocol when run.

II Input Validation

In both the ATM and BANK components, we developed custom functions to validate user input, ensuring adherence to the correct protocol and preventing integer overflow. We implemented many checks on each parameter to ensure the input conforms to the desired format, leveraging use of regular expressions. The following listed are the custom functions we created for input validation. Note that each function returns a value of 0 if it fails to meet the format specifications and 1 if it adheres to the specified format.

`int username_check(char *username)`

The username is limited to a maximum of 250 characters, exclusively consisting of alphabetical characters without any symbols or numbers. We used the regex pattern `/^[a-zA-Z]+$/` to enforce this constraint.

`int amount_check(char *amt)` - similar to deposit_check & balance_check in bank.c

The amount is a positive integer with a limit set to the maximum possible integer value in C, without surpassing this upper bound. We used the regex pattern `/^[0-9]+$/` to verify that each character in the string is a digit. The string is then converted to a long and compared to the maximum integer value.

`int pin_check(char *pin)`

The pin is a 4-digit number. We used the regex pattern `/^[0-9]{4}$/` to verify the string contained only 4 digits.

III BANK

The bank program creates and stores user information. Through the bank, users can make deposits and check their balance. Moreover, when the ATM tries to check and modify bank records, the bank also validates user input when interacting with the ATM. Two lists were added to the Bank structure, with usernames being mapped to both pins and balances. Bank can be divided into two parts: the local side that provides an interactive place for users to create their account, check their balance and deposit money. The other side is the remote side that reacts and sets up communication between an atm and the bank record, as explained in Section V of the document. To process commands in the bank, the entire command-line string is tokenized. In doing so, the program can validate the number of parameters inputted one by one. It's important to note that if the sole input is the Enter key or a

newline character, the program simply continues and prompts for input again. Here are the following commands that a user can use to interact with the bank directly:

`create-user <user-name> <pin> <balance>`

This command can be broken into 4 tokens. The first token must match `create-user` and only 3 tokens should follow that contain information. If a 5th token exists in the command, then the `Usage:  create-user <user-name> <pin> <balance>` prints to the Bank command-line. The validation functions from Section II are used to match the constraints with each parameter. Once each parameter is checked, we check if the username exists in the list. If it does not, then we create a new user entry in the list. Otherwise, no changes are made to the list since the user already exists.

`deposit <user-name> <amt>`

This command is broken into 3 tokens. The first token must match `deposit` for the deposit function to begin. A similar process is repeated to validate the username and deposit amount. Note that was also checked if the amount deposited will cause the total to be larger than the maximum integer size, returning an error in that case.

`balance <user-name>`

This command is only broken into 2 tokens. The first token must match `balance` in order for us to retrieve the balance information. The username is validated and then we check the bank to make sure the user exists.

IV ATM

ATM is a program that is able to connect to the Bank via the router, which is what allows bank and ATM to send and receive information between each other. ATM prompts the user to start a session, and in order to verify that the user is who they are, they need to enter the pin associated with their account from when they created their account in the bank to verify their identity.

In the ATM struct, we introduced 3 supplementary variables. The current user that is logged-in is stored in `atm->username`. This helps identify who is logged in, preventing unauthorized access within the same session. Additionally, `atm->max_tries` stores the maximum allowable number of failed login

attempts. Finally, a list called `atm->pin_tries` keeps track of each user's unsuccessful number of attempts. Once a user reaches their maximum allowable failed login attempts, their account is locked and the ATM prevents the user from logging in again. The ATM supports the following commands:

`begin-session <user-name>`

This command only has 2 tokens, and the first token must match `begin-session` in order for the command logic to continue. We check if the username is valid based on the constraints in Section II. We then send this information to the bank to check if the user exists, and returns an error otherwise. Then, we prompt the user to enter their associated PIN with the account. We allow a maximum of 5 tries before the user account is locked. Once they are in the session, the user can run the rest of the commands to their account. If a user inputs this command while already in a session, then an error statement is printed and no changes are made.

`withdraw <amt>`

This command only has 2 tokens, and the first token must match `withdraw` in order for the command logic to continue. Using the validation function, we check if the amount is a valid amount to withdraw from the bank account. If the bank account has enough money to withdraw the given amount, then the command dispenses the money and changes the balance in the bank. Otherwise, the program lets the user know that there are insufficient funds and prevents the amount from being withdrawn.

`balance`

This command only has one token, and the one token must match `balance` in order for the command logic to continue. It checks the bank, finds the user and their associated balance, and gives back the amount stored in the bank as long as the account is still logged in.

`end-session`

The command only has one token, and the one token must match `end-session` in order for the command logic to continue. If the user is logged in, then the username stored in the atm is then removed and the user must log back in. If that no one is logged in, then a error returns telling the user that no user is logged in

V BANK AND ATM COMMUNICATION

When a user is created in the Bank, a `<username>.card` file is created. This file holds the user's pin. When a user wants to log in, the atm looks for an existing `.card` file to authenticate that the file has not been tampered or created by a user by comparing the content inside the file. Since we know that all `.card` file should hold the corresponding user's pin, we will verify the content of the .card file before allowing user's even attempt to enter their pin for safety measures. The pin authentication is done by the ATM sending a message to BANK regarding a user. The BANK will either deny this request or send a message back to the ATM with the encrypted user's pin. If the ATM request gets denied then the ATM will prompt "No such User". If the ATM receives a message back from Bank with the user's encrypted pin, the ATM will decrypt the pin then prompt the user to enter a pin to check if the pin matches the actual pin that we received from Bank. Once the user is verified, then the ATM can start sending commands to the Bank.

In order for the ATM to communicate to the Bank and make changes to a user account, we have to build commands that are processable by the bank program and point to the right side-effect necessary. The ATM and Bank support 3 connections to retrieve and modify data within the bank: search for a user, withdraw money from a user account and check balance on a user account. For each ATM input, we validate the input using the functions from Section II and then build the commands in accordance with the table below:

ATM Command Message	Success Response from Bank	Fail Response from Bank
search <username>	<encryptedPin>	DNE
withdraw <username> <amt>	richboy	brokeboy
balance <username>	<userbalance>	no response returned

Mitigated Vulnerabilities

I. Preventing Stack-Based Buffer Overflow Attacks

Buffer overflow occurs when an attacker attempts to exploit weaknesses in how the program handles and manages memory, especially in the stack. Attackers can inject their own executable code or retrieve memory addresses of certain variables in the stack by using format strings. Additionally, manipulating input in a way that exceeds buffer limits can induce errors, further compromising the integrity of the stack. This can lead to unintended program behaviors that can put user accounts at severe risk.

In order to prevent this, we implemented strict input validation to prevent attackers from manipulating the commands to add format specifiers like `%p` which can read memory addresses. We included the `regex.h` library and used regex commands to validate our input. Here are examples of some of the regex we used:

Lines 259-282 in bank.c

```
int username_check(char *username) {
    …
    reg = regcomp(&regex, "^[a-zA-Z]+$", REG_EXTENDED);
    if (reg == 0) {
        valid = regexec(&regex, username, 0, NULL, 0);
        regfree(&regex);
        if (valid == 0) {
            return 1;
        }
    }
    return 0;
}
```

Lines 232-249 in atm.c

```
int pin_check(char *pin) {
    …
    reg = regcomp(&regex, "^[0-9]{4}$", REG_EXTENDED);
    if (reg == 0) {
        valid = regexec(&regex, pin, 0, NULL, 0);
        regfree(&regex);
        if (valid == 0) {
            return 1;
        }
    }
    return 0;
}
```

As you can see here, each of these functions correspond to different parameters in each of the commands used on the interface for Bank and ATM. In order to ensure strict pattern policies, we were very explicit in the regex we used. For username, we first checked to make sure that the string itself was within 250 characters. We then used the regex pattern `"^[a-zA-Z]+$"` which basically says match the string if it only contains uppercase or lowercase alphabet letters. Everything else is excluded so string formatters with a `%` in the beginning will not compile in the regex. The validation for the pin is similar, but instead the `"^[0-9]{4}$"` regex pattern will only compile if the string strictly has 4 digits. Doing this ensures that the input adheres to the predefined formats, reducing the risk of buffer overflows and other related issues.

Additionally, we incorporated memory-safe functions such as `strncpy` and `strncat`, enhancing security by allocating a predetermined amount of memory for specific operations.

Line 292, 305 in atm.c

```
strncat(cmd, username, sizeof(username));
strncpy(filename, username, strlen(username));
```

Their counterparts—`strcpy` and `strcat`—dynamically adjust memory allocation based on perceived needs. The use of these functions help prevent buffer overflow by restricting the amount of data that can be written to a buffer, thereby mitigating the risk of stack corruption and associated attacks.

II. Brute Force Attack with Username and 4-Digit-Pin

Since the pin for a user is only 4-digit long, there are 10,000 possible combinations that an attacker would have to cycle through using a script to try all the combinations. This is assuming the attacker has a list of usernames or obtained a username through other means.

On the ATM side, we implemented a maximum number of allowable unsuccessful pin entries. If a user fails to enter their pin 5 times, then the account will remain locked, and that user would essentially have to go to a system administrator to unlock their account again. The chances of guessing the pin significantly decreased, making it harder for attackers to break in.

Lines 53-54 in atm.c

```
atm->max_tries = 5;
atm->pin_tries = list_create();
```

Lines 274-285 in atm.c

```
int curr_tries = list_find(atm->pin_tries, username);
if (curr_tries == -1){
      list_add(atm->pin_tries, username, 0);
      curr_tries = 0;
} else {
      if (curr_tries >= atm->max_tries) {
            printf("Account Locked: Too Many Failed Attempts\n");
            return;
      }
}
```

Lines 345-354 in atm.c

```
if (count != 4) {
      list_change(atm->pin_tries, username, curr_tries + 1);
      printf("Not authorized\n");
} else {
      if(pin_check(pin) == 0 || strcmp(decryptpin, pin) != 0) {
            list_change(atm->pin_tries, username, curr_tries + 1);
           printf("Not authorized\n");
…
```

In our code, we created a local variable to capture the current number of failed login attempts by the user. If this number is -1, it means the user hasn't entered their PIN incorrectly before, so we add a new entry for them and start keeping track of failed attempts. If a number is retrieved, we then check if the current attempts equal the maximum allowed. If true, the program displays an error message. This list is updated when prompting the user to enter a PIN. If the entered PIN is incorrect, the program adjusts the list and increments the attempts by 1.

III. Session Hijacking when User is Inactive

We cannot rely on users to consistently practice good security measures, such as logging out after completing their session. Leaving a session open creates a vulnerability since attackers can exploit it and gain control. To reduce the risk of this happening, we set a timer that starts right after the user's last interaction with the system. If there is no command input within 3 minutes, the session is automatically terminated. However, if a user enters another command within this timeframe, then the

timer resets, which prevents unnecessary session terminations and maintains a good user experience while still ensuring security.

In atm-main.c

```
fd_set rfds;
struct timeval time;
while (1) {
      FD_ZERO(&rfds);
      FD_SET(0, &rfds);
      time.tv_sec = 180;
      time.tv_usec = 0;
      check = select(1, &rfds, NULL, NULL, &time);

      if (check == -1) {
            exit(0);
      } else if (check) {
            // process command
      } else {
            if(atm->username != NULL) {
                  …
                  printf("Timeout - Inactivity for 3 minutes\n");
                  atm_process_command(atm, commandend);
                  …
            }
      }
}
```

We set up a loop to continuously check for user input. It sets a timer for 3 minutes and waits for either user input or the timeout. If there's input, it processes the command; otherwise, if the timeout occurs and the user is logged in, it triggers a timeout message and processes the `end-session` command. The program remains in this loop until it exits by these means.

IV. File Tampering Attack - Checking contents of card to make sure data integrity is main

File Tampering Attacks occur in the case when a cybercriminal creates their own <username>.card file of a user they're trying to hack into or modify an existing <username>.card file. Therefore Data Integrity is important in this program, we need to maintain all data being stored anywhere in the program remain untouched, in such cases of an attacker trying to modify a user's password and login as such user.

When a user is created in BANK, we create a unique .card file associated with that user and the content stored inside the .card file is the user's pin. When we validate a user login in atm, we check to see if the content of the .card file is the same as the one the bank created when create-user is called, and not modified in any way by an attacker. This security measure basically serves as an authentication measure and prevents attackers from creating their own .card file and using it as if it was one that the bank created.

Lines 304-327 of atm.c

```
FILE *card = fopen(filename, "r");
if (card == NULL) {
      printf("Unable to access %s's card\n", username);
      return;
}
char cardcontent[5];
fgets(cardcontent, sizeof(cardcontent), card);
…
if (strcmp(cardcontent, decryptpin) != 0) {
      printf("TAMPERED CARD FILE (CONTENT DONT MATCH) !!!!\n");
      return;
}
```

V. Man-In-The-Middle Attack

Storing sensitive information such as the PIN is dangerous because this makes it vulnerable to interception attacks if the attacker is able to access our network or files. If the attacker is able to intercept the PIN through the network, they will know the PIN and access the user's account. When the atm grabs the user's PIN information from bank.c, an attacker may be able to intercept that information in order to steal the user's pin in order to login themselves later. In order to mitigate this from happening, when the atm requests bank.c for the user's PIN number to validate user input, the bank encrypts the pin number using the encryption function from OpenSSL. It then sends this encrypted pin to atm.c for it to decrypt and user later on after the user enters a PIN when requested. After the atm.c file confirms that the .card file has not been tampered with and the user enters a valid PIN format, the atm program compares decrypted version of the PIN from the bank file and user inputted PIN to see if they entered the right PIN number. This strategy adds an additional security

measure aimed at thwarting Man-in-the-Middle Attacks. In the event that a cybercriminal intercepts traffic between the BANK program and ATM program, decryption of the message would be necessary for successful hacking of our system.

Lines 202-210 of bank.c

```
char pin_str[5];
snprintf(pin_str, 5, "%d", pin);
unsigned char cipherpin[128];
unsigned char *unsignedPin = (unsigned char *)pin_str;
int cipherpin_len;
cipherpin_len = encrypt(unsignedPin, strlen ((char *)unsignedPin), key,
iv, cipherpin);
bank_send(bank, cipherpin, cipherpin_len);
```

Lines 319-322 of atm.c

```
unsigned char decryptpin[128];
int decryptlen;
decryptlen = decrypt(response, idx, key, iv, decryptpin);
decryptpin[decryptlen] = '\0';
```

Other Considerations

Replay Attacks and Eavesdropping

When the attacker gains access to the router, the attacker could intercept and retransmit valid data previously exchanged between bank and atm to perform unauthorized actions to gain knowledge on the data exchange protocol. Information that the attacker could intercept include the encrypted PINs, ATM commands, and bank responses. However, since we have implemented end-to-end encryption features for the transmission of the PIN, where bank encrypts the PIN and sends the ciphertext to the ATM to be encrypted, the attacker will not have access to the plaintext of the PIN, which makes it more difficult for the attacker to gain access to crucial information.

Denial-of-Service Attacks

If the attacker were to flood the network with a high volume of traffic by creating and sending new packets to both ATM and bank to overwhelm the system's resources, the ATM services would be impossible for the users

to access due to long processing time or the exhaustion of the system's resources. A similar attack can occur when the attacker drops packets to be exchanged between ATM and bank, causing incomplete transactions and dysfunctional applications.

SQL Injection

We thought about making some sort of SQL Injection vulnerabilities, however since this program does not use any sort of SQL database. We were unable to integrate any SQL Injection defense tactic as there is no need for it.