ATM Project Report

Alex Neumyer, David Kombe, Hai Le, Martin Iglesias
Computer and Network Security

May 6, 2020

Contents

§1 Protocol

Our protocol consist of 4 programs:

1. "`BANK`": Command-line interface used by bank manager to set up and manage customers accounts.
2. "`ATM`": Command-line interface that is used by a bank customer to perform operations securely within their bank accounts.
3. "`ROUTER`": Enables "`ATM`" and "`BANK`" to communicate with each other.
4. "`INIT`": Enables "`ATM`" and "`BANK`" to communicate securely by establishing an initial shared secret key.

§1.1 Protocol Flow

First the `ROUTER` program will run, enabling communication flow between `ATM` and `BANK`. Then, `INIT` will run. Using a python script, we pre-computed 100 random (p, g) pairs, where p is a random 8-digit prime, and g is a primitive root of p. The `INIT` program randomly chooses one of these pairs. The values p and g, along with a random initial key k for encryption, are stored in the files `<file_name>.bank` and `<file_name>.atm`. The file name and path are inputted to the `INIT` program. `ATM` and `BANK` will then read these values line by line; ensuring the same values of g and p for the Diffie-Hellman Key Exchange operation as well as the same initial k value for encryption and decryption.

Afterwards, `BANK` will run and read p, g, and k from its initialization file. `BANK` will show a command-line interface through which the bank manager can: create a user, deposit money into a specified user's account, and check a user's balance. All user information is stored in a linked list within `BANK`. Every linked list element contains the fields:

<username, balance, SHA256(PIN), SHA256(username+SHA256(Pin)+timestamp), next_user>.

The user PINs are sent and stored as hashes, and thus the PINs themselves are never known by the `BANK` program. Furthermore, `BANK` creates a card file of the format <username>.card. `BANK` also stores the SHA256(username+SHA256(Pin)+timestamp) into the card file and its database for future comparison purposes. The `BANK` remains running waiting for incoming connections.

A bank customer can then access their account by running the `ATM` program. `ATM` will read the g, p, and k from its initialization file. Immediately, it will show a command-line interface through which the customer can begin a session as a customer. Once a session has begun, the authenticated customer can withdraw money, check their balance, and end their session.

Note:

1. At the beginning of each command, `ATM` and `BANK` perform a Diffie-Hellman Key Exchange using the shared g and p values. This allows them to securely acquire the same key for encryption and decryption purposes. Any message sent from `ATM` to `BANK` or vice versa is encrypted/decrypted with the newly generated key.
2. If at any point, the protocol encounters a non-positive feedback (incorrect pin, too much money to withdraw, etc.) both `ATM` and `BANK` come back to their idle states and the corresponding error message is printed at the `ATM` interface.

§1.2 Begin-session Protocol

The bank customer inputs begin-session <user-name>. `ATM` will then send a payload to bank to check if the user exists in the `BANK` database. The format of the payload is:

[command_len][check_username_remote][usename_len][username]

`BANK` will receive this payload and search its database for the given user name.`BANK` will then send 1 if the username exists, and 0 other wise.

`ATM` waits for a response from `BANK`. If the received payload indicates a valid user, the `ATM` will request the user's PIN, which is obfuscated in the command-line by "*" characters. When the PIN is given, `ATM` asks `BANK` if the given PIN is correct. Instead of sending the value of the PIN, `ATM` will send the hash of the PIN to `BANK` as:

[command_len][check-pin-remote][SHA256(pin)]

`BANK` will receive this payload and make sure the SHA256(PIN) that was stored with the customer's user name matches the SHA256(PIN) received. Upon a successful match, `BANK` will read from the card file and make sure the content of the username.card matches the content within its database. It will reply back with 1 if hash matches or 0 otherwise.

Finally, `ATM` receive payload. If the payload indicates a valid pin, `ATM` will let the user log in. The prompt will then be changed to ATM(user). The withdraw and balance operations will then be available to the user.

§1.3 Withdraw Protocol

User inputs withdraw <amt>. `ATM` will then ask the `BANK` if that amount is valid (if there is enough money in the customer's account to withdraw that amount). Format of the payload:

[command_len][withdraw-remote][username_len][username][amt]

`BANK` will receive this payload and search its database for the balance of the given username. If the withdraw amount requested is valid, `BANK` sends a 1 to `ATM`, 0 otherwise.`ATM` will then receive the response from `BANK`. If the response is valid, it will display the requested amount.

§1.4 Balance Protocol

User inputs balance. `ATM` will then ask bank for the user's balance with the following payload:

[command_len][balance-remote][username_len][username]

`BANK` will receive this payload and look for the username's balance in its database. It answers back to `ATM` with the following payload:

[balance]

ATM read the balance received and prints it out to the command-line interface.

§1.5 End Session Protocol

User inputs end-session. ATM logs the user out and clears the state of the logged in variable. The prompt then changes.

§2 Potential Attacks and Defense

§2.1 Pin Read From ATM and BANK Transmission

§2.1.1 Attack Synopsis

Pin could be read from transmission between `ATM` and `BANK` or could be printed using a format string vulnerability.

§2.1.2 Defense

PINs are hashed on the `ATM` side when entered and transmitted in hashed form. Furthermore, PINs are hashed on the `BANK` side when a user is created and saved in the List as a hash. If the hash is somehow made available to the attacker, they will not be able to figure out the pin by any means other than brute-forcing all possible PIN hashes and comparing.

§2.2 Buffer Overflow

§2.2.1 Attack Synopsis

An attacker could exploit a buffer overflow vulnerability in our code. They could have the program display the contents of the buffer that holds the key used by `ATM/BANK` during program execution. If the attacker is able to successfully exploit this vulnerability, they can use that key to transmit properly encrypted messages or decrypt messages from either `BANK` or `ATM`.

§2.2.2 Defense

In order to defend against this, we made sure that there are no buffer overflow vulnerabilities in our code. We used `strncat()` and `strncpy()` instead of `strcat()` and `strcpy()`. By doing so, we specify the amount that we want to be copied into the desired buffer. Furthermore, we changed/adjusted our buffers to correct sizes to prevent unused space.

§2.3 Attacker Intercepting Messages

§2.3.1 Attack Synopsis

An attacker could intercept our messages and know its formats. With that information, they could create their own packets; thus send those spoofed packets to `BANK` pretending to be `ATM`, and vice versa. For example, the attacker could see the format of our `BANK ->ATM` withdraw protocol's as:

[command_len][withdraw-remote][username_len][usename][amt]

By knowing the payload of the format, the attacker could send the following payload to `BANK`:

[15][withdraw-remote][6][martin][10000000000000000000000]

The attacker can then sends the success payload to `BANK` ([1]).Upon receiving a success payload, `ATM` would dispense the requested amount.

§2.3.2 Defense

We implemented an encryption and decryption protocol using AES-CBC 256 (similar to the Cryptography Project). The encryption and decryption process will use the key received by the Diffie-Hellman Key Exchange protocol. By encrypting all of our payloads with an unknown key, even if the attacker intercepts our payloads they will not be able to uncover the plain text of the payload.

§2.4 Brute force PINs

§2.4.1 Attack Synopsis

The attacker could implement a script and brute force their way to figuring out the PIN of a specific user.

§2.4.2 Defense

We made sure the attacker couldn't brute force the PINs by implementing a time-out protocol. Any user can only attempt to begin-session for a specified bank customer at most 3 times. After the third failed attempt, there will be a time out of 60 seconds before that user could try again. The time out also increases by 5 minutes with every set of 3 failed attempts.

§2.5 Spoof Success Payload

§2.5.1 Attack Synopsis

The attacker could login as a valid user with X amount of money, and withdraw a **valid** amount of money. Then they could inspect the packets sent from `ATM` to `BANK` and vice versa. Since the attacker withdrew a valid amount of money, the packet sent from `BANK` to `ATM` will be the our success payload (an **encrypted 1**). With this information, the attack could withdraw $1,000,000, drop the packet from `BANK` to `ATM` and just replicate the the success payload. `ATM` would receive a 1 indicating a valid operation, thus withdrawing the $1,000,000.

§2.5.2 Defense

We defended against this attack by generating a new key for every transmission between `ATM` and `BANK`. By doing so, our encrypted success message will always differ due to the difference in key when encrypting.

§2.6 Attacker Peeking/Adjusting Card File

§2.6.1 Attack Synopsis

The attacker can either have the content of the card file or the user's PIN. If our card files are generated with no contents inside, there are two possible scenarios:

1. Attacker has the PIN but not the card's content:
    a) Upon receiving the correct PIN from the user, `ATM` just check if <username>.card exists without checking the content within that card file. By knowing that information, despite not having the card file, the attacker could simply create an empty card file with that username, and thus successfully log in. Furthermore, even if the card file holds a hashed PIN or PIN+username, the attacker could figure out the hash by trial and error.
2. Attacker has the card content but not the PIN:
    a) If the content of the card file is plain trivial information like the PIN in plain text, the attacker could just use that PIN to log in.

§2.6.2 Defense

We defended against this attack by filling the card file with a security hash. Our security hash consists of the SHA256 of the combined string of username, hashed pin, and a timestamp of this hash is computed. By including the time stamp, even if the attacker has the PIN and username, the attacker cannot run SHA256 and hash it themselves.

§2.7 Attacker Peeking User's PIN

§2.7.1 Attack Synopsis

While the user is entering in their PIN, the attacker could implement a program in the host's machine and peek the user's PIN.

§2.7.2 Defense

We defended against this attack by hiding the PIN as the user types it into the `ATM` command-line interface. Then we replace the characters of the entered PIN with "*". Therefore, even if the attacker could "see" the user enter in their PIN, they won't know what it is.

§3 Unimplemented Potential Attacks

§3.1 ATM Drop Connection Attack

§3.1.1 Attack Synopsis

An attacker could potentially attack the `ATM` system by dropping the connections between `ATM` and `BANK`. This could be a Denial of Service attack, in which the user cannot access their account, nor perform any `ATM` operations. Furthermore, if the `BANK` already deducted X amount of money from the account, the `ATM` won't receive the success payload due to the loss of connection. Therefore, the `ATM` will never dispense the money to the user, and thus `BANK` will simply lose X amount of money.

§3.2 Key Logger Attacks

§3.2.1 Attack Synopsis

An attacker could potentially implement a key logger program within the host's machine. By doing so, they will gain the host's `ATM` credentials and successfully perform any `ATM` operations.

§3.3 Unauthenticated Payloads

§3.3.1 Attack Synopsis

Since the payloads are not authenticated using an HMAC or another form of digital signature, anyone in the network could potentially send a payload to `BANK` or `ATM`. Furthermore, because these payloads are not authenticated, an attacker could attempt to flip a bit in a transmission and possibly (although extremely unlikely) change the amount of money withdrawn or send success when the operation was actually invalid.

§3.4 Multiple ATM Instances

§3.4.1 Attack Synopsis

An attacker could potentially start multiple instances of `ATM` and connect them simultaneously to the `BANK`. By doing so, our program would break.