CMSC414 ATM Project: Build It

Design Report

Group 30

Suhani Chopra, Grace Tao, Kelly Kim, Mia Fetter, Smrithi Chakravarthy

Overall Protocol Design

Introduction

Our goal for this project is to create a system that facilitates communication between a bank, ATM, and users to simulate real-world financial transactions with consideration for potential adversaries. Our main programs for functionality include:

- **Init:** establishes an initial bank state and creates files
- **Encryption:** contains code shared between the bank and ATM for encryption purposes
- **Bank:** stores and manages account information
- **ATM:** allows users to interact with banking information

Sensitive information is passed between entities through a router, so we must ensure that the data remains protected against unauthorized attackers or other threats in the network, and that only certain authorized data gets passed between the bank and ATM.

In this report, we will demonstrate our usage of encryption methods and other security features to maintain a functional, secure environment for a bank.

Init

In order to establish a new session, we must run the **init** and **router** programs to generate initial files and start the router, respectively.

The **init.c** program takes in one command line argument:

```
<path1>/init <path2>/<init-fname>
```

This creates two files in the specified directory:

```
<path2>/<init-fname>.bank
<path2>/<init-fname>.atm
```

The implementation allows users to choose their own filepath. The two .bank and .atm files are used separately for running the bank and ATM programs. Each of these now stores two things: (1) 32-byte AES key to encrypt pin numbers, and (2) a different 32-byte AES key concatenated

to encrypt messages sent between the bank and atm. These keys are the same between the files. We decided to use separate keys for separate purposes so that if an attacker has access to one key but not the other, they are still somewhat restricted in the information gained and malicious activity enacted. Since we assume attackers cannot access the .bank and .atm files, attackers will not be able to feasibly decrypt any pins or messages since they don't have access to the encryption keys. The behavior of the init program works as follows:

- If a user fails to provide precisely one argument, then print "Usage: init <filename>" and return value 62
- If the .atm or .bank files already exist, print "Error: one of the files already exists" and return value 63 without writing over or creating either file
- If the program fails for other reasons, print "Error: creating initialization files" and return value 64
- If the initialization is successful, print "Successfully initialized bank state" and return value 0

Encryption

The encryption folder contains the main program **enc.c**, and a header file **enc.h**. In the encryption program, we implemented functions to encrypt pin numbers and messages, as well as a function to decrypt messages sent between the bank and ATM.

- **enc.c**: main file containing encryption functions
    - **generate_rand_bytes():** generates the keys and IVs used for encryption
    - **encrypt()**: uses AES-256-CBC for encrypting pins
    - **gcm_encrypt()**: uses AES-256-GCM for encrypting messages between bank and ATM, with a unique 16-byte tag
    - **gcm_decrypt()**: uses AES-256-GCM to decrypt encoded messages sent to ATM from bank. If the extracted authentication tag differs from that created by gcm_encrypt(), the program terminates

Bank

The bank.c program maintains a Bank struct, which contains (1) the path to the .bank file, and (2) a list of User structs, each containing a username and their current balance. The list is set to null each time a new bank session begins, and cleared each time a session ends. Only users created during the session are added to the list. To begin a new bank session, the bank program takes in one command line argument using the file created from init:

```
<path1>/bank <path2>/<init-fname>.bank
```

If the .bank file provided in the command line argument cannot be opened, the bank prints "Error opening bank initialization file" and returns value 64. Given a valid .bank file, the bank can process the following three commands (any others print "Invalid command"):

1. `create-user <username> <pin> <balance>`

   We first check that the command is valid (has 4 arguments, username is all alphabetic characters and less than 250 characters, pin is 4 digits, balance is nonnegative and less than INT_MAX) and that the user doesn't already exist.
   Then, a .card file is created for the user. The contents of the .card file are as follows:

   ```
   [encrypted pin][user-specific initialization vector]
   ```

   The user's pin is encrypted using AES-256 cipher block chaining mode of operation to ensure pin storage security. A unique, random 16-byte initialization vector is also generated and stored alongside the encrypted pin to ensure that two identical pins don't encrypt to the same ciphertext. The key used for encryption is the first 32 bytes of the .bank file.

2. `deposit <username> <amt>`

   If the user exists in the users list and the inputs of this command are valid (has 3 arguments, user exists, amount deposited is nonnegative and does not cause integer overflow) then the user's balance is updated with addition of the specified amount.

3. `balance <username>`

   If the user exists in the users list and the inputs of this command are valid (has 2 arguments, user exists) then the user's current balance is printed.

ATM

The atm.c program allows for users to access or modify certain banking information from an outside source (i.e. an ATM machine). The program maintains an ATM struct that tracks whether or not someone is currently logged in, who the current user is (if anyone), and a list of login attempts for each username. To begin a new ATM session, the program takes in one command line argument using the file created from init:

```
<path1>/atm <path2>/<init-fname>.atm
```

If the .atm file provided in the command line argument cannot be opened, the bank prints “Error opening ATM initialization file” and returns value 64. Given a valid .atm file, the ATM can process the following four commands (any others print “Invalid command”). The `begin-session`, `withdraw`, and `balance` commands are sent to the bank, where the bank extracts user information to send back to the ATM. ATM receives and echoes back the responses.

1. `begin-session <username>`

   To ensure that the user actually exists in the bank’s users list, we send over this command as an encrypted message to the bank. The bank maintains a list of all users created in the current session and if <username> is not found, then the ATM prints “No such user”. This is to prevent attackers from posing as users by creating .card files that might happen to have the same file name as an existing bank user. If the user exists, the ATM prompts them to enter their pin number. The pin they enter is first encrypted using the first AES key in the .atm file and stored IV in the .card file, then checked against the stored encrypted pin in the .card file. If the two encryptions match, this means the pin is correct and the user is logged in to their account. The prompt then changes to “ATM (<username>): “ to signify this. The user is given five attempts to enter the correct pin; if all attempts fail, then the user is locked out of their account for the remainder of the session.

2. `withdraw <amt>`

   If a user is logged in, the inputs of this command are valid (2 arguments, amount is non negative and does not cause integer overflow), and the user’s balance has sufficient funds, then the user’s balance is updated with subtracting the specified amount.

3. `balance`

   If a user is logged in, and the inputs of this command are valid (one argument), then print the user's balance (information is drawn from the bank).

4. `end-session`

   If a user is logged in, and the inputs of this command are valid (one argument), then terminate the current session and log the user out, reverting the prompt back to "ATM: ".

ATM-Bank Communication

The `begin-session`, `withdraw`, and `balance` commands inside atm are sent to the bank to execute, since only the bank tracks user creation and balances. Below is the two-way communication protocol to send messages along the insecure UDP router:

1. ATM sends an encrypted message *C* to the bank to execute
2. Bank decrypts *C* and executes the ATM's command
3. Bank sends an encrypted response *C'* to the ATM
4. ATM decrypts *C'* and prints the bank's response

Messages sent between the ATM and bank are formatted like so:

```
[encrypted msg length][encrypted msg][init vector][authentication tag]
```

We utilize functions from enc.c to ensure that all communication between the two entities remain encrypted.

Messages are encrypted through AES-256-GCM mode. The second of the AES keys found in the .bank and .atm files is the one used for encryption. Details about this encryption scheme and why it's ideal are found in the countermeasures section below.

Attacks and Countermeasures

1. **Brute force attack on pins**
    a. We have implemented a limit of five login attempts for each username. This prevents attackers who want to potentially gain access to account information by guessing the pin by repeatedly entering different pins, since there are only 10^4 possible pins. Our mechanism locks users out of the account if they have surpassed five attempts at entering the pin into the account. Even if subsequent attempts are correct, the user will still be locked out of the account until the session ends.
    b. This protection against a brute force attack is in atm.c in the atm_process_command function. In this function, the number of login attempts by a user are counted and if they exceed MAX_ATTEMPTS, then they are locked out from starting a session.

2. **Stealing pin information from a user**
    a. We have prevented malicious users from stealing pin information that is stored on cards through encryption. Instead of storing pins as numbers or plaintext which attackers can easily use if accessed, we have encrypted the pins as described previously, with the encryption key securely stored in the .bank and .atm files. Even if attackers do somehow get access to card files, they won't be able to actually use it because it is cipher text.
    b. The 256 bit AES key that we are using for encryption is stored in the .bank file created by init.c. Specifically, we are using AES in CBC (cipher block chaining) mode. To make the pin encryption unique for each user since we are only employing this one key in the .bank file, we are also concatenating randomly generated initialization vectors (IV), using them for the encryption with the key, and storing them with the encrypted pin in the card files.

3. **Man in the middle attacks**
    a. Ensuring the security of the remote commands between the bank and the ATM is very important, as it is susceptible to man in the middle attacks. Attackers could potentially intercept the communication between the bank and the ATM, giving

them the ability to read commands, modify withdrawals or deposits, change usernames and balances, and other harmful behaviours. To prevent this, we have encrypted the messages between the atm and bank with an AES-256-GCM algorithm.

b. The AES-256-GCM algorithm ensures confidentiality and authenticity, meaning even if messages are somehow intercepted, the attacker (1) cannot read what the message is saying, and (2) will be found out if they try to modify the message. The IV adds randomness to the encrypted message to ensure that attackers cannot derive meaningful patterns in any intercepted messages; i.e. they cannot learn the format of how a begin-session, withdraw, or balance command is encrypted. AES-GCM also generates a unique authentication tag. The tag is 16 bytes long, and if the attacker tries to modify the encrypted data, the tag verification will fail because you cannot have a valid tag without the key.

c. Every time gcm_encrypt() inside enc.c is called, the protocol itself generates a new 16-byte authentication tag, which is the scheme's method of knowing whether or not a message comes from a trusted source. When a bank receives a message, gcm_decrypt() is called. The decryption scheme checks if the authentication tag came from the gcm_encrypt() function; if it did, then we know the ATM sent the message and we proceed as normal. If the tag source isn't recognized, then we do not proceed and terminate the bank program as we think an attacker has modified the message. For example, if in atm we just generated a random 16-byte tag and sent it to the bank, gcm_decrypt() would not recognize it as the one it created, and error. In our code, the bank program will terminate immediately if gcm_decrypt() returns -1, i.e. tag verification failed.

4. Pattern analysis or recognition

a. Attackers could potentially try and see if there are patterns in the pin and message encryption in an effort to determine the types of messages being sent or what the pin numbers are.

b. Both of our encryption methods have unique and randomly generated initialization vectors that will add randomness to ciphertexts such that no two identical pins or messages encrypt to the same ciphertext. For pins, this means

that even if an attacker knew person A's pin, it is hard for them to know who else has the same pin, preventing them from gaining entry into the account of anyone besides person A. For messages, initialization vectors decrease the chance of noticeable patterns, which is crucial because each command type will be formatted the same way before being encrypted (e.g. withdraw is always "withdraw <username> <amt>"). Not being able to tell which type of command is represented by the message prevents attackers from sending messages of similar formats that the bank interprets as authentic.

5. **Buffer overflow vulnerabilities**
    a. There is potential for a buffer overflow attack in the system when the user inputs a command. They could try to send a longer command to overwrite memory that has important information as we have them stored in arrays. Also, the buffers that receive the messages between bank and atm could possibly exceed the buffer size to override authentication tags that we have as a part of our encryption process.
    b. Our code is secure against these types of attacks through our safe practices of string functions, buffer checks for commands, input validation, and null termination, and freeing allocated memory.
        i. **Safe practices of string functions:** We made sure to use strncpy instead of strcpy because of its explicit size limits, and snprintf instead of sprintf. This enables safe string manipulation; for example, when we process commands in atm_process command in atm.c, we keep an original copy of the command and process the copy with strtok and null termination. This prevents the strtok function from reading past the buffer.
        ii. **Buffer checks for commands**: if the command entered by the user exceeds the buffer size, we print an error to stderr and the command is not processed.
        iii. **Input validation:** we made sure to validate numeric input when we use strtol so that the converted integer won't exceed INT_MAX. This way, attackers can't input an arbitrarily large number to compromise memory.
        iv. **Null termination:** when we write contents into a buffer, we make sure to have checks that null terminates the string and also make sure that we do

not exceed the limits of the buffer. This way they can't be exploited by an attacker to reach other parts of memory. For example, in atm.c when we make copies of commands for processing, we add null termination and enforce buffer limits like command_copy[sizeof(command_copy) - 1] = '\0';

v. **Freeing allocated memory:** when we allocate memory for various data structures, we free the memory after use preventing attackers from exploiting dangling pointers to overwrite memory. For example after we are done sending and verifying received messages we free the message after we send it to bank in atm.c ("free(sendline)) so it's no longer accessible

Other potential attacks

- **Denial of service attack:** An attacker could attempt to login multiple times over and over again, which would make the memory grow. Even if the ATM blocks a user from doing too many attempts, previous attempts are stored, so an attacker could keep trying to login, ultimately overwhelming the system. This could be mitigated by carefully studying system capabilities and ensure the allowed number of attempts would not hinder the application's functionality

- **Encryption time analysis:** An attacker could keep track of how long the encryption process is for a specific pin, which may get them closer to discovering what the pin is. Constant-time cryptographic operations could be used to mitigate this to ensure that encryption always takes the same amount of time regardless of the input, preventing attackers from inferring information based on timing. If this is not possible, a time delay could be integrated into response time to ensure that response time does not provide any clues that could be utilized toward pin discovery.

- **Social engineering attack:** Attackers may manipulate users into revealing their PINs or sensitive information through phishing or impersonation. This is beyond the scope of this project but could be mitigated through educating users on recognizing phishing attempts, safe ATM usage practices, and adding an additional layer of authentication in the form of 2-step verification.

- **Replay attack:** An attacker could potentially intercept valid data transmissions. For example, the user could enter their pin and the attacker could later resend ("replay") this transmitted data to gain unauthorized access. Since the ATM system may not detect that the request is being replayed, the attacker could gain access to the system as if they were the legitimate user. One measure to mitigate against replay attacks could be the addition of timestamps and a corresponding acceptable window to processed requests. This would deem a replayed request invalid because it would not meet the set time constraints. Otherwise, we could track the ciphertexts that have come in, and if one is repeated (i.e. same exact encryption despite the use of IVs which almost guarantees uniqueness), then we terminate.