Team: Slobodan Simic & Michael Brown

1. Defense in depth

The main majority of the project focuses around handling valid input. However even after the direct line of input it is important to check input whenever it is used to further ensure that initial checks are not bypassed. Throughout the code we attempted to follow this practice and ensure valid input whenever it is called. One of the most used examples of this in our code is the use of strtok function which helped us decide if the given input format was correct by checking the number of tokens in the input and seeing if it corresponds to the number of tokens we are expecting. We also used memcpy with a specified length to try to prevent possible buffer overflows.

2. Use of Canary

In the functions atm_process_command, bank_process_local_command, and bank_process_remote_command I included a char canary called getting. Throughout each if statement in each method, along with the necessary check I did to ensure I would proceed, I also checked that my Canary value was not changed. This helped us with checking for potential buffer overflows from a malicious user.

3. Chosen plain text

Although we were not able to implement it secret keys are a great way to protect data over the network. Since the attackers in this case can perform chosen-plaintext attacks it is important to use randomized and unknown keys. One way to do this is in the init file when initializing both the .bank and .atm files generate a random key of substantial size and store in both files. You could then use this value as a one-time pad to encrypt data and generate a new key using the first key as the randomization seed. This way you can perform a one-time pad and generate a new key to continue to simulate one-time pad while distributing the key through the file which the attack cannot access.

4. Brute force

One attack is to continually guess random pins for user cards. This is a way to brute force in or possible disrupt services by overflowing the atm or bank. We did not implement this but it would have entailed locking out a user if they have incorrectly guessed a user and pin pair a set number of times.

5. Buffer overflow

One way an attacker could try and exploit the program is through the use of extremely large inputs to override parts of the program especially since stack guard is disabled. Throughout this project we used a variety of methods to prevent such an attack by ensuring fixed sized operations on all data and memory modifications.

6. Audit log

One way to prevent future attacks is to have a detailed summary of transactions. This feature would help to reveal the presence of an attack. Although we were not able to implement this it would involve creating an audit log file that contained every transaction and the time it occurred.