Design Documentation

Our safety protocols begin with the init file format as we ensure that the file path is legitimate and can be found. Then for our bank and atm, they communicate over the router, which leaves a vulnerability when someone can see our message. So we encrypt and decrypt our plaintext messages between the atm and the bank. Furthermore, the bank process requires a encrypted message to run, so if a person controlling the router would send a previous encrypted message to the bank, to run an illegal operation, then it would not decrypt correctly because the key generation is time based, thus preventing the attack. We also prevented a buffer overflow by making sure that the command has a valid amount of data before we work with it. We also set a limit to the amount of users being created, because we did not want an overflow of the bank.

Listed below are potential attacks against our implementation that, if executed as instructed, are effectively countered by our system.

(i) Server-side Buffer Overflow

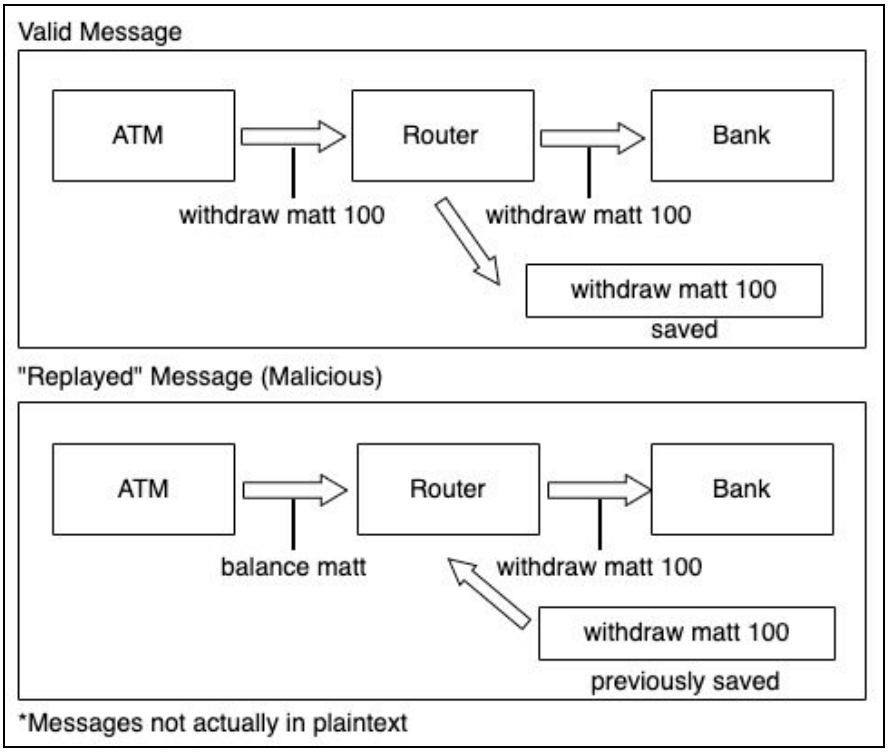
When sending and receiving information from the server, it is possible for adversaries to give a specially crafted input which can include a combination of a noop sled and injected shell code that will write to memory outside of the contents of the buffer. The buffer overflow fix that we implemented was that every time that we receive information from the router, we ensure that a valid amount of data is placed into a separate, safe buffer which further characters beyond expected input won't be read from. If you were to compromise the router and send a buffer overflow attack to the bank (where all of the valuable information is stored), our program would be able to prevent this attack.

(ii) Client-side Buffer Overflow

Just as before, our program anticipates that malicious input in the form of overloaded input that may occur from the command line as well. This would essentially occur when an adversary attempts to type out an excessively large command in the CLI that would in theory overflow the buffer we use to process the command. However, we foil this threat by carefully reading only a specified amount of bytes (chars) from the command line at a single time and if the command meets or exceeds a threshold (the maximum valid command length) then we flag the rest of that input as invalid and clear it out of the command line before taking another request. We handle it this way to avoid excessively large inputs that attempt to crash our system or cause unexpected functions.

(iii) Replay Attack

In the scope of this project a “replay attack” refers to the fraudulent re-use of valid commands to maliciously exploit our system’s functionality. In a simplified sense, this is started by first monitoring messages sent over the router (from the atm to the bank or vice versa) and storing them for later re-use before passing them to their respective destination. After valid messages are stored, the malicious adversary will then wait for another message to come to the router in the same direction and replace that message with the previously stored message in order to maliciously re-produce an earlier result. This process is depicted below,

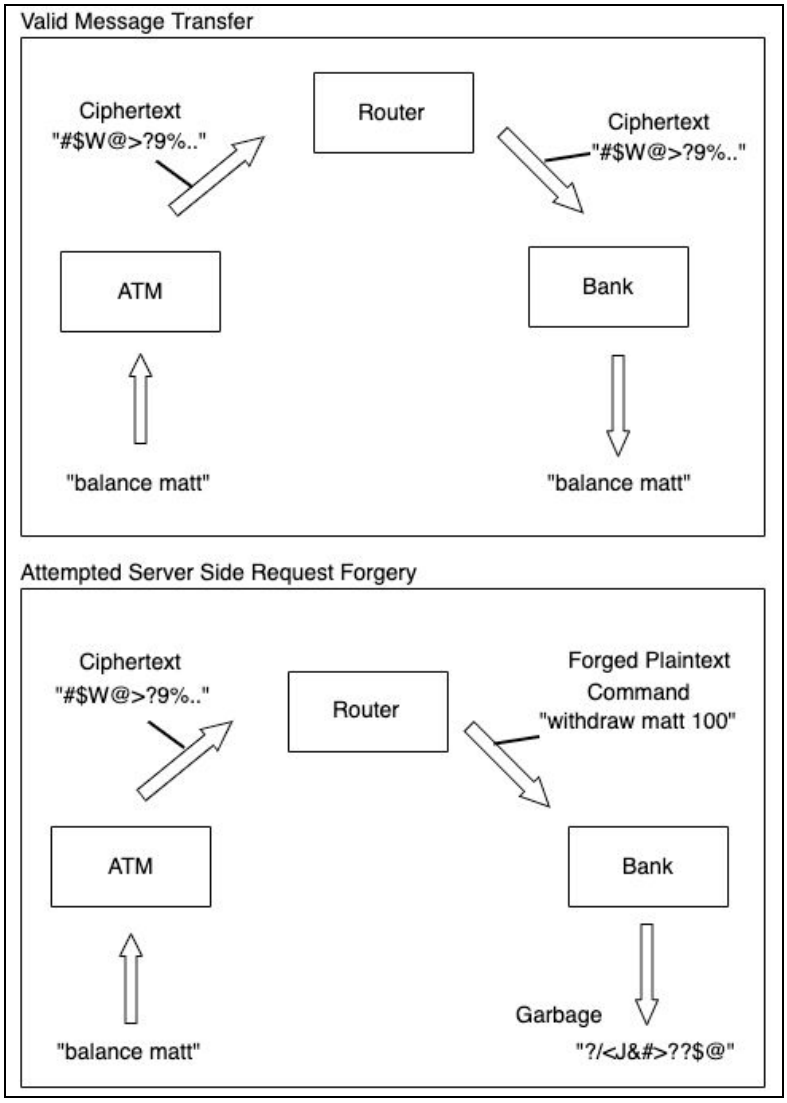


In our implementation, we counter this attack method by first utilizing the time that our program is initialized as a seed to create an actual seed that will be used later for encrypted message transfer later. This seed is then written into the initialization files for the bank and atm where it is extracted by the main methods for atm and bank respectively. The seed is then used to initialize respective random number generators for the atm and bank. As these random number generators for the atm and the bank are seeded identically, the random number generators are synchronized. From this point on, the random number generators are called to produce 32 byte keys every time either participant (the atm and bank) sends or receives data that is used to encrypt or decrypt the messages transferred over the router. As these keys are synchronously generated

in the atm and the bank, there is no need to send it over the router where it can be intercepted or read. We do this approach because if we were to merely use static keys (with a static IV) our messages would be encrypted but constant across repeated messages. For instance, sending “withdraw matt 100” once would render some encrypted ciphertext that would be the same if that exact same message were to be transmitted again. By using synchronously generated random keys, the actual ciphertexts for duplicate messages are different and therefore not eligible to be replayed because decrypting a replayed message produces garbage that will not be processed.

(iv) Server-side Request Forgery

The bank/atm implementation that we created for this project involves messages being passed across a communication router. In order to protect all of the requests across the server, it is important to encrypt communications between them. We implemented a two-way AES256 symmetric CBC encryption that encrypts messages in both the ATM and the bank before sending over the information. On the receiving end of the message, the messages are decrypted using the same key and IV information which are required to encrypt/decrypt the message. As described in the previous section, the same key is randomly generated, using a common random generator seed on both ends, every time in order for the encryption/decryption information to remain constant for each while providing a randomization layer which makes figuring out the messages exponentially harder. These measures are taken in order to prevent any eavesdropper on the router from figuring out what the messages are. If an active eavesdropper had the knowledge of the plaintext information, it would be easy for them to intercept the raw data and give back the other side what it wants to know. In this case, the hacker would have full control over messages going across the router. They would be able to tell the bank and atm anything that they want to access and be able to use potentially sensitive information to their advantage. This protects the users from having their pin and balance being known to a hacker while also preventing them from figuring out authentication information giving them full access to the account without actually knowing the pin. For example, if a user tries to log in with their pin, a hacker would be able to figure out the pin and gain unwanted access to the account. On the other side if a hacker was able to see the information the bank sends that authenticates the login, they could pass the authentication info back to the atm and get access without ever needing to know the pin.



(v) User Overflow

Another attack that a malicious user may attempt is creating so many users that the hash table cannot function and our program crashes. Our protocol combats this by comparing the number of users added to the number of bins we specify upon creation of our hashtable each time the user attempts to create a new user. If the number of users already contained in the hashtable is equal to the number of bins in the hashtable we do not allow the user to create any more users. We tested this by temporarily reducing our hashtable size to 3 bins and then creating 3 users. On the attempt of creating a 4th user, access was denied. Once we saw that the protocol correctly prevented any more create-user commands we decided that 200 users would be an adequate max number of users.

(vi) Command Injection

Command injection is when a malformed argument is provided when executing a program to run commands they weren't supposed to. An attack that you can do when running is to provide a filename that has harmful characters. Our code is vulnerable to command injection as that seems beyond the scope of our code. It didn't matter if we sanitized the argv for init because it would not even run our code. So if you had a single quote in your file name then you will get this terminal. However, if we were able to access the code of argv[1] then we would have sanitized the output and not allow harmful characters. In init we treat the single argument as a string and we separate the path and the file name between two strings and we make sure that the path exists. Then we concat the path with the filename that they provided. Also, there is no system call within the code that would be vulnerable to a command injection. Also, our executables will not run if you did not provide enough or provided too much arguments. We also test if the path you provide has at least one forward slash, and if not then that is an error.

(vii) Format String Vulnerabilities

Within our implementations of the bank and the atm, we treat user input carefully so that it is never mistakenly run as code. In essence, we are preventing any opportunity for an adversary to utilize our programs with invalid commands that could perform some (potentially dangerous) function on the backend. For instance, if a user were to input '%p' at any point in either of our prompts it would never replace those characters with a memory address as it would if it were run as code.