

# CMSC414: Homework 2

by Michael DeWitt and Tyler Dunn  
March 11, 2012

## Protocol

Our protocol works in several steps, each of which is necessary to provide the strongest security possible in the system. It is reasonable to assume that a bank has the capability to store and keep track of secure keys to interact with each of its ATMs. To this end, we decided that the best approach would be to set up a system of shared private keys between the bank and the ATM itself - in our system, there are two separate private key-pairs, each of which is known only by the bank and the machine itself. Users have no interaction with the keys, as all operations involving the keys are isolated in the bank and the ATM.

These two keys are used for separate purposes - one for AES and one to calculate an HMAC tag for the messages which we pass. Separation of the keys roles is important, as it is often the case that otherwise-secure cryptography is substantially weakened by the reuse of keys for multiple purposes. Each of our shared private keys only needs to be generated once (perhaps during the initial programming of the ATM), with one key from each pair getting stored on the machine and a copy of each respective key stored in the bank. Once the key pairs are created through the use of a cryptographically-secure pseudorandom number generator, they get hard-coded into the respective systems that they secure. Neither of these keys need to change during the lifetime of the ATM, unless there is a physical compromise of the machine (this idea is expanded upon in the Attacks section).

Any message which is passed over the network by either party follows a very specific structure:

- First, messages are encrypted with the shared AES key. This key was limited to 128 bits by US Export Control restrictions (at least, we believe that's the root of the error that we got when trying to increase the size of the key to 256 bits). To ensure that our messages were as securely encrypted as possible, we chose to make use of the chain block cipher (CBC) mode of encryption with AES. This was facilitated by a cryptographically-secure, randomly generated initialization vector and PKCS5 padding to fill out block lengths. The Java Cryptography Extension provided the necessary utilities for the padding and AES encryption in both the ATM and the bank's software. The results of the encryption (subsequently referred to as the ciphertext) are considered to be cryptographically secure, meaning that no one without the secret key can tell what was encrypted with greater than  $[\epsilon+1/2]$  probability, where  $\epsilon$  is a negligibly small number.
- Next, the current POSIX time (in milliseconds) is prepended to the ciphertext, in order to associate it with a particular point in time. This step becomes important later, in message verification.
- Finally, the message composed of the [time, ciphertext] pair is passed through a hash-and-MAC operation to generate a tag. Our 256-bit HMAC key is used to perform this step. The result of this step is a message-tag pair, which associates our ciphertext/timestamp with a tag generated by the ATM. This ensures that, regardless of what happens to the message or the tag, we can be confident that we will know whether or not any of the contents have changed. This also allows for authentication, since only the ATM and the bank will be able to sign the messages with the corresponding tags.

# Verification of Message

When the bank receives any message from the ATM, it is vetted to make sure that it is a valid message. Remember that any message the bank receives will be of the form [time, encrypted message, tag]. Specifically, the bank goes through several steps to ensure its validity:

**MAC** The bank makes sure that the [time, encrypted message] pair hashes to the [tag]. If it does not, then the message has been tampered with and the requested command is not valid.

**Reasonable Timing** The bank checks that the message was sent a reasonable amount of time ago. We decided to hard-code this threshold to be ten seconds for the purpose of this project. Should the [time] portion of the message reflect that the message was sent more than ten seconds ago, we decide that there is a high likelihood that the message is an attempt at a replay attack.

**New Message** The bank makes sure that this message is new. What this means is that this is the first time the bank has seen the message. If any message has arrived bearing a timestamp more recent than this message, there is a high likelihood that this message is an attempt at a replay attack. Therefore, the bank will reject any message that was sent at the same time or before any previously sent message. Additionally, this step verifies that no valid command is accidentally executed twice.

Once a message passes these three steps, the bank accepts the input and valid and decrypts the message to begin processing.

## Specific Protocols

There are a number of times that the ATM has to communicate with the bank to carry out its various tasks. Then, the bank has to respond.

### Authorization of a user at the ATM to the bank:

ATM: begin-session Alice  
PIN? 0000

The ATM verifies that the user's card is present and reads in the value in the file. Then, it prompts the user for a PIN, which it verifies to be 4 numeric digits. Passing those tests, the ATM sends a message to the bank in the form: `AUTH: [user] : [PIN] : [card_value]`

If the provided PIN and `card_value` match the values in `[user]`'s bank account, then the user is verified. Should everything check out, the bank sends the message `AUTH: [user] :PASS` or, otherwise, `AUTH: [user] :FAIL`.

### User checking balance at the ATM

ATM (Alice): balance

The ATM sends `BALANCE: [user]` to the bank, and waits for the bank to respond with `BALANCE: [user] : [amount]`.

### User withdrawing money at the ATM

ATM (Alice): withdraw 100

If the user provides both the command withdraw and a value to withdraw, the ATM sends the bank a `WITHDRAW:[user]:[amount]` message. Should the user have enough funds, the bank sends back `WITHDRAW:[user]:SUCCEED`. If the user does not have enough funds, the bank responds with `WITHDRAW:[user]:INSUFFICIENT`. If there was something wrong with the input, the bank sends `WITHDRAW:[user]:FAIL`.

## Attacks and Vulnerabilities

Since this protocol is intended to be secure, it should be resistant and resilient to as much tampering as possible, in addition to providing secrecy. The following list consists of a variety of diverse (but common) attacks, and describes how our protocol deals with each:

### Client-impersonation/Server-impersonation

This attack is infeasible, since it is computationally difficult to pass meaningful messages as either end of this system without knowing the cryptographic keys used to construct them. The bank will not respond to messages which are meaningless, and vice-versa.

### Chosen-plaintext (and known-plaintext)

AES is not vulnerable to chosen-plaintext attacks, and neither is the SHA1 hashing used for HMAC. Even if users could present their own plaintexts, which they cannot, this would not allow them to know anything about the other data encrypted with that key.

### Chosen-ciphertext

AES is not vulnerable to chosen-ciphertext attacks, a condition which also holds for SHA1. Even though ciphertexts can be passed to either end of the system, it is inconsequential, since there is a very low likelihood that valid messages can be reconstructed from a randomly-chosen ciphertext.

### Man-in-the-Middle

Since we make use of pre-shared keys for encryption and message-authentication, we can avoid any potential man-in-the-middle attacks. Any changes to the message are going to be flagged as being invalid (since no one but the valid parties can construct the associated hash-and-MAC tags). Similarly, since chosen-ciphertext/chosen-plaintext attacks and replay attacks are taken care of, we can be reasonably assured that there are no valid Man-in-the-Middle attacks to worry about.

### Overflow

All input is sanitized and checked for validity before it is allowed to interact with parts of the system. For example, users are not allowed to withdraw or deposit negative amounts of money, have more than `Integer.MAX_VALUE` money in their account, and the user's name/card/PIN are not allowed to contain characters which could break the delimiters in the ciphertext or the message. Buffer overflow attacks were deemed to be outside the scope of this assignment.

### Replay

Since the POSIX time is incorporated into each of the messages, it is not feasible for any replay attacks to take place. Both systems check to ensure that the time of each subsequent request is greater than the ones prior to it. Additionally, since requests expire after ten seconds, it is difficult to replay requests in a meaningful timeframe anyway.

### **Brute-force**

The weakest part of this system is the low-entropy PIN, which can be brute-forced in a very reasonable amount of time (since there are only 10,000 possible values to be associated with a given username/card combination). We decided that the brute-forcing of PINs was outside the scope of this project, since those types of attacks are not considered to be exploits by the grading criteria.

### **Denial-of-Service (DOS)**

Any compromised router can decide to arbitrarily drop packets, so we were not overly concerned about DOS attacks. The situation in which an adversary is directly sending malicious or malformed data is taken care of by the HMAC. If anything is awry in a message, it gets discarded.

## **Results and Conclusion**

Despite rigorous testing, we were unable to find any true exploits in the system. All of the attacks listed above are inconsequential or are beyond the scope of this project. We chose to remove the keys from the source code of the program after it was compiled - this is to remove any possibility of people learning the keys by simply accessing the raw source files. We believe that we've designed is secure, and the software that we've written implements it correctly.