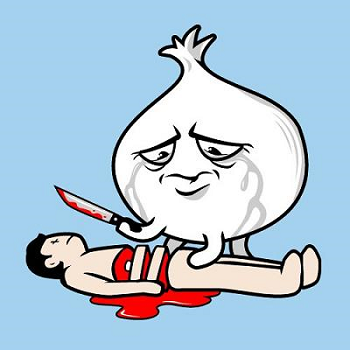
**By: William Shi, Ayane Naito, Laszlo Kopits, Allan Tenenbaum, Jocelyn M., Marie Tessier**



**Table of Contents**

1. **eCTF Challenge**

a**.** Initial Instructions

b. Scoring Guide and Point Checklist

c. Sequence Diagrams

i. The Onion

ii. The Funyun

1. **The Original Onion**
   1. **ATM to Bank Communication**
      1. Established an AES tunnel between the ATM and Bank using Diffie Hellman.
         1. Diffie Hellman is done every time the system is booted/power-cycled.
         2. All subsequent communication between the ATM and Bank is encrypted with AES.
   2. **Card to ATM Communication**
      1. While the ATM and Bank have an AES tunnel, the card and the ATM do not. Therefore, all data sent between the card and the ATM is encrypted (different methods depending on the process).
   3. **RSA “Onion”**
      1. A two-layer RSA encryption, where both the bank and the card must decrypt one layer each to access the data inside.
      2. Used to encrypt the balance
      3. Process
         1. The bank sends the two-layer RSA-encrypted “Onion” to the ATM, which sends it to the card after decrypting the AES.
         2. The card decrypts the outer layer of RSA before sending it back to the ATM, which encrypts it with AES and sends it to the bank.
         3. The bank decrypts the AES and removes the inner layer to read the balance.
            1. This is so the balance cannot be modified by the user without the bank knowing, as the bank knows the balance before the card.
         4. The bank encrypts the balance with the RSA key that the card has the corresponding private key to, and sends the balance to the ATM.
         5. The ATM decrypts the AES and sends the RSA-encrypted balance to the card.
         6. The card decrypts the RSA and receives the balance.
2. **Deprecated RSA(Card, ATM, Bank)**
3. Card
   1. Three letter for loop command line
   2. RSA decrypt functionality(BearSSL)
   3. RSA signature(BearSSL)
4. ATM
   1. The ATM has an AES key that encrypts and decrypts AES encryption between the ATM and the bank.
      1. It cannot decrypt the AES encryption between the bank and the card.
   2. Sends requests and packages to and from the bank and card.
5. Bank
   1. Database

-JSON file including contents that will be provided to attacking teams.

- Where keys will be stored, and the encryption used within the database

* 1. Communication

-Initial Diffie Hellman Key exchange to establish the AES-256 tunnel.

-. AES-256 Tunnel specifications.

1. Tears :the death of the onion
   1. Time constraints
   2. Possible communication issues
2. **The Funyun**
3. ATM to Bank Communication
   * 1. Established an AES tunnel between the ATM and Bank using Diffie Hellman. \*If time not left to be implemented, pre-loaded keys.
        1. Diffie Hellman is done every time the system is booted/power-cycled.
        2. All subsequent communication between the ATM and Bank is encrypted with AES.
   1. Card to ATM Communication
      1. While the ATM and Bank have an AES tunnel, the card and the ATM do not. Therefore, all data sent between the card and the ATM is encrypted (different methods depending on the process).
   2. AES “Funyun”
      1. Instead of a two-layer RSA encryption, the Funyun uses AES with CBC encrypted balance and IV to transfer to the card and decrypt.
      2. Process
         1. The bank sends the AES-encrypted “Funyun” to the ATM, which sends it to the card.
         2. The card decrypts the AES and sends it to the ATM for the next transaction key.
         3. The ATM sends it to the bank, which will generate a new passkey for the balance.
         4. The ATM will be sent the user’s balance, so that the user may check/withdraw money.
4. **Bank**

a. Database

i. JSON file including contents that will be provided to attacking teams.

ii. Where keys will be stored, and the encryption used within the database

b. Communication

i. Initial Diffie Hellman Key exchange to establish the AES-256 tunnel.

ii. AES-256 Tunnel specifications.

1. **ATM**
   1. Communication
      1. The ATM has an AES key that encrypts and decrypts AES encryption between the ATM and the bank.
         1. It cannot decrypt the AES encryption between the bank and the card.
      2. Sends requests and packages to and from the bank and card.
2. **Card**
3. Design
   1. Receive three letter function commands from atm(continuously looped)
   2. Standardized flash storage for all key parameters.
4. Provisioning
   1. Storage of all production variables
5. For Loop Command Prompt
   1. Three letter functions and their usage
6. Functions
   1. Additional Functionals added
   2. External function usage notes

**8. Acknowledgements**

1. BWSI
2. MITRE instructors

**Section 1: The eCTF Challenge**

There are 3 components:

\* Bank with database of accounts

\* ATMs distributed around area with cash

\* Users with ATM cards who know their PIN

To withdraw cash, a user must insert the original ATM card issued by the bank and then enter a 4 digit PIN. Only after successfully completing this, the user may then choose one of three options:

\* Read their balance

\* Change their PIN

\* Withdraw an amount of cash from the ATM less than or equal to their balance

**Attacks to Prevent Against**

Misuse by legitimate users:

\* An individual should not be able to withdraw more money than is in their account

Fraud by illegitimate users:

\* An ATM card thief should not be able to access the account without the PIN

\* A snooper who sees the PIN should not be able to access the account without the card

\* Someone with access to the card should not be able to clone the contents and create a duplicate card

Snooping by hackers:

\* An attacker skimming (recording) the traffic between the ATM and bank should not be able to access the account

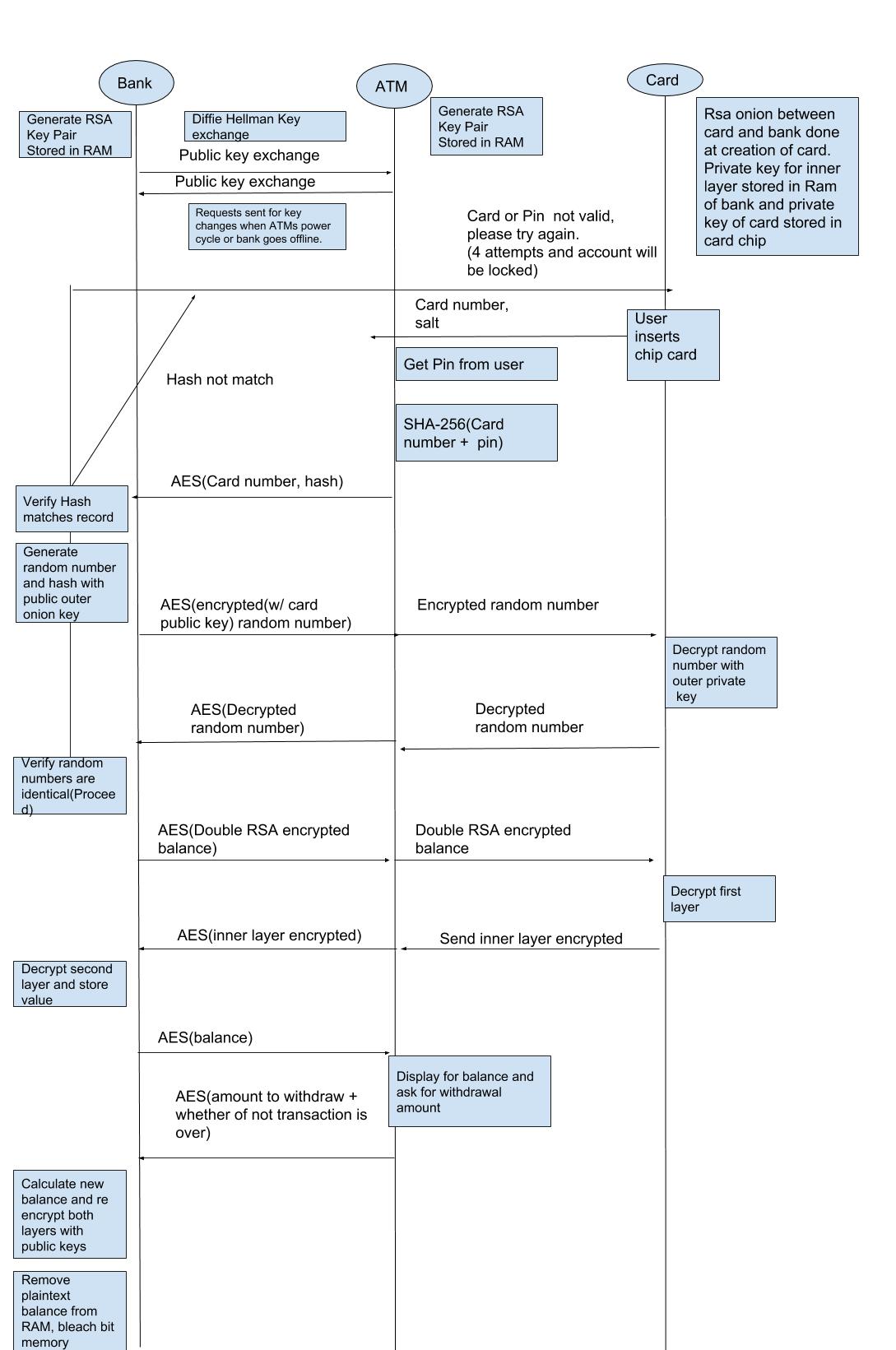
\* An attacker skimming the traffic between the ATM and ATM card should not be able to access the account

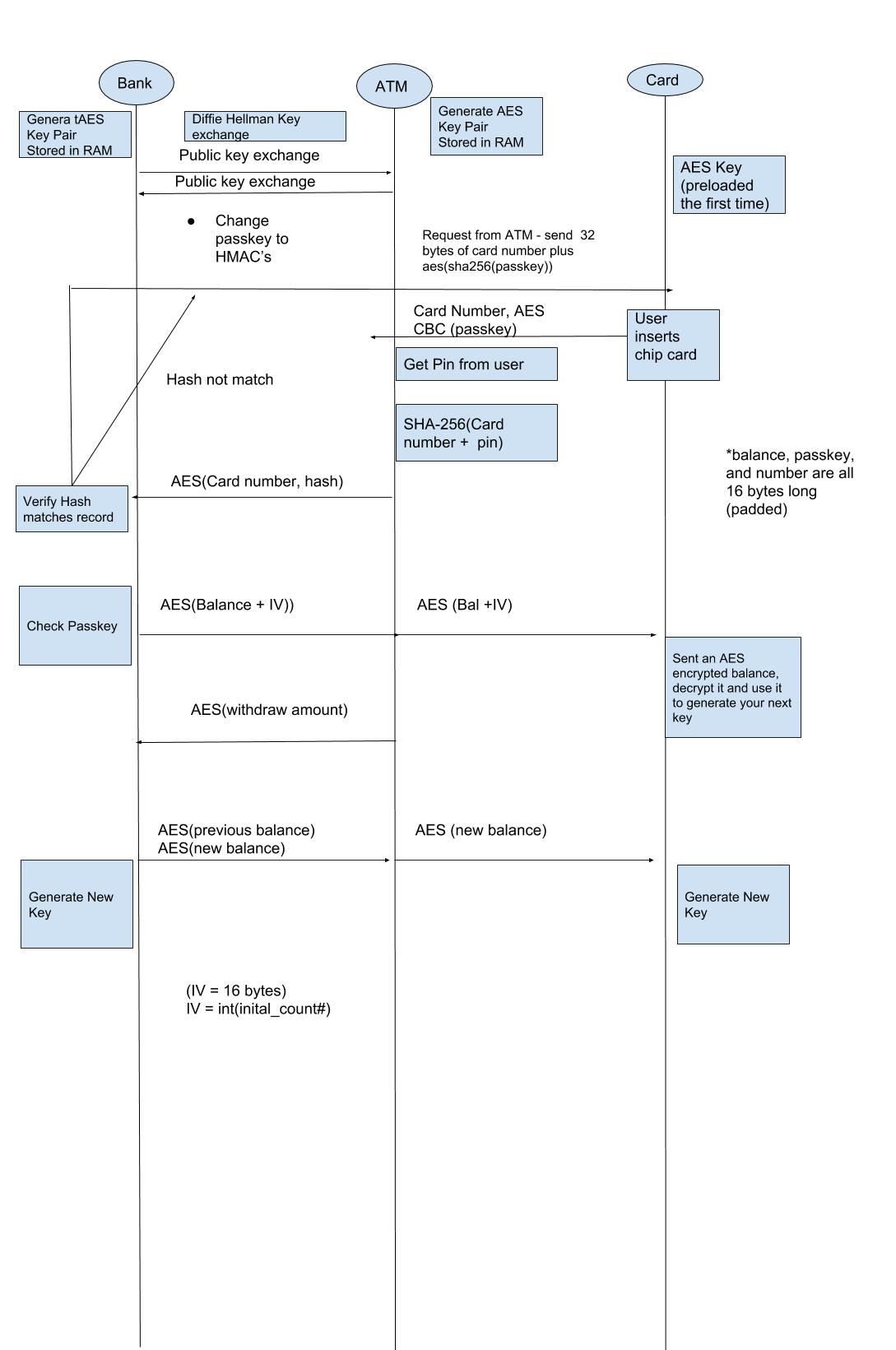
\* If the bank database is leaked, no PINs or balances should be compromised

**Challenge**

Design a high level protocol to securely provide these services. If encryption is used, explain how keys are distributed and stored.

Scoring rules for the ATM-Bank challenge is as follows (points from the door lock challenge carry over):  
  
Submitting a design = 500  
  
Documentation:  
0 - Intentionally obfuscated code  
50 - Poor to no in-line comments, no high level documentation  
100 - Decent in-line comments, no high level documentation  
150 - Decent to go in-line comments, minimal high level documentation  
200 - Excellent in-line comments; logical, easy to follow code; good high level documentation  
250 - Excellent documentation throughout \*CAN BE GIVEN TO AT MOST ONE TEAM\*  
Attack points:  
Capturing a flag = 50 + 25 if you're the first team to capture it + 50 if you're the only team to capture it by the end  
List of flags:  
Database Breach  
 1. Find correct PIN  
 2. Find correct Balance  
ATM  
 3. Withdraw more than 10 bills from any ATM (your "own card" will be able to withdraw 10 bills)  
Own card  
 4. Withdraw more than 10 bills from your own account  
Stolen Card  
 5. View the balance  
 6. Change the balance  
PIN only - no card  
 7. View the balance  
 8. Change the balance  
Skimmed card  
 9. View the balance  
 10. Change the balance  
Cloned card (clone your "own card" in any means possible, return it to us, and make a transaction with the new card)  
 11. View the balance  
 12. Change the balance  
Defensive points:  
25/unattacked flag/day

**Sequence Diagrams: Onion followed by Funyunz:**



**Section 2: The Original Onion**

**ATM to Bank Communication**

Diffie Hellman Key Exchange

To keep communications secure for the customer and the bank, a Diffie Hellman key exchange is necessary for every session that the customer opens with the bank. All of the transactions that go from ATM to Bank will be encrypted with an AES 256 secure tunnel.

* The Bank will have its own private keys AES, RSA (per each ATM) generated and stored in RAM.
* Every time that the ATM power cycles, it will generate its own private keys and store it in RAM (per machine, not per card). A request will be sent to the database to change the key corresponding in the bank database in RAM.
* When the bank powercycles, ATM communications will be cut and unavailable. The bank will send requests to re-generate keys to the ATMs. Communications will only be able to unlock until both parties send an encrypted confirmation message to each other.
* The public keys are generated and will be stored in non-volatile memory.
* Diffie Hellman is implemented within bank.py and atm.py bank-side and ATM-side. In the case that the testing renders that it not feasible, keys will be preloaded.

Card to ATM Communications

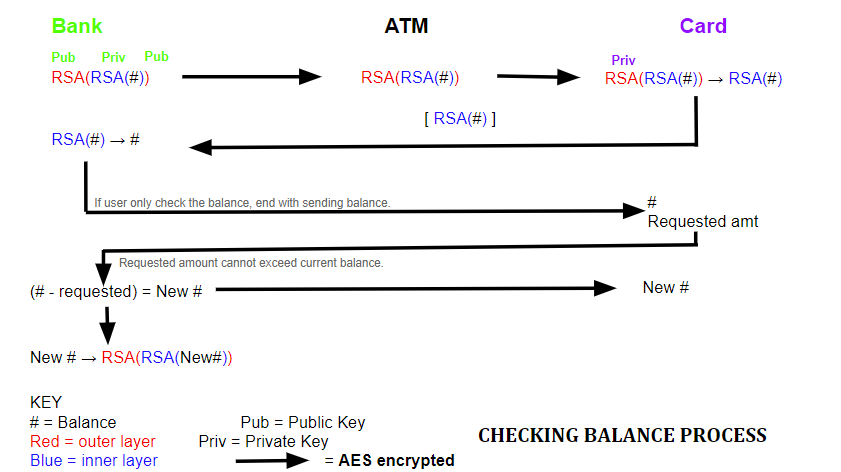
While the ATM and Bank have an AES tunnel, the card and the ATM do not. Therefore, all data sent between the card and the ATM is encrypted. Within the Onion, the only things transferred over the card is as follows:

* Initial Card Number to the ATM
* The Onion: [(Balance)RSA 1]RSA0
* Various Authentication Steps (see sequence diagram: Onion)

RSA Onion

The Onion is a two-layer RSA encryption, where both the bank and the card must decrypt one layer each to access the data inside. It is used to encrypt the balance and process user transactions of withdrawal and checking the balance.

1. The bank sends the two-layer RSA-encrypted “Onion” to the ATM, which sends it to the card after decrypting the AES.
2. The card decrypts the outer layer of RSA before sending it back to the ATM, which encrypts it with AES and sends it to the bank.
3. The bank decrypts the AES and removes the inner layer to read the balance. This is so the balance cannot be modified by the user without the bank knowing, as the bank knows the balance before the card.
4. The bank encrypts the balance with the RSA key that the card has the corresponding private key to, and sends the balance to the ATM.



**Section 3: The Funyun**

ATM to Bank Communication

Diffie Hellman Key Exchange

Much like the Onion, the Funyun uses the same initial process of a Diffie Hellman key exchange to swap the symmetric 256-bit keys. By doing this it keeps communications secure for the customer and the bank. The process is same for the Funyun as it is the Onion, except the main difference is RSA has been completely dropped from the Funyun.

* The Bank will have its own private keys AES CBC (per each ATM) generated and swapped. Only after the exchange has happened will sessions be available to open.
* Every time that the ATM power cycles, it will generate its own keys and store it. A request will be sent to the database to change the key corresponding in the bank database in RAM.
* Diffie Hellman is implemented within bank.py and atm.py bank-side and ATM-side. In the case that the testing renders that it not feasible, keys will be preloaded.

Functions located in Bank.py for Immediate reference:

# Generates the modulus and base for Diffie Hellman using a prime number

def diffie\_hellman(self)

modulus = self.generate\_prime\_number(2)

base = self.generate\_prime\_number(3)

return (modulus, base)

# Bank-side diffie hellman function, which sends the modulus and base to ATM before computing agreed value.

def diffie\_bank(self):

mod, base = self.diffie\_hellman()

# Sends modulus and base to ATM

self.default\_write(struct.pack(">32s256s256s", format("dif\_mod\_base"), format(mod, 256), format(base, 256)))

secret\_number\_b = random.randint(1, 9999)

side\_bank = (base\*\*secret\_number\_b) % mod

# Sends bank's half of diffie hellman to ATM.

self.default\_write(struct.pack(">32s256s", format("dif\_side\_bank"), format(side\_bank, 256)))

# Receives ATM's half of diffie hellman from ATM to compute final value.

the\_size = self.bytesize(side\_bank)

print "The size: ", the\_size

transaction\_id, side\_atm = struct.unpack("32s256s", self.default\_read(288))

print(self.bytesize(side\_atm))

# uptime\_key\_bank is the final bank-side agreed value for diffie hellman

self.uptime\_key\_bank = (side\_atm\*\*secret\_number\_b) % mod

# Links commands in ATM-Bank interface to functions in the bank

# Three letter codes link interface commands to bank functions.

# Initializes AES Key first upon power cycle.

Card to ATM Communication

While the ATM and Bank have an AES tunnel, the card and the ATM do not. Therefore, all data sent between the card and the ATM is encrypted.

* Initially, the card will send over a 32-bit AES passkey with the Card ID
* The card will have to decrypt the AES Funyun balance with the passkey on the card.
* New passkeys will be generated for each session.

AES “Funyun”

Instead of a two-layer RSA encryption, the Funyun uses AES with CBC encrypted balance and IV to transfer to the card and decrypt.

1. The bank sends the AES-encrypted “Funyun” to the ATM, which sends it to the card.
2. The card decrypts the AES and sends it to the ATM for the next transaction key.
3. The ATM sends it to the bank, which will generate a new passkey for the balance.
4. The ATM will be sent the user’s balance, so that the user may check/withdraw money.

Fun Fact: Why is it called the Funyun?

* **Funyuns** is the brand name of an onion-flavored corn snack originally from 1969. Funyuns consist primarily of cornmeal A salt and onion mix gives them their flavor, but it does not contain any onion.
* This name very well reflects the nature of the Funyun Protocol. You’d like to think it is the Onion, and has the spirit of the Onion too. Unfortunately, it does not contain the actual Onion, but something that replicates the taste of the Onion. Through the replication of the Fake Onion “Funyun”, it gives us something more feasible to implement.

**Section 5: The Bank**

**Database**

Everything that needs to be kept, information wise, will be stored in a JSON data file that will be initialized by db.py. Within the database a list of crucial information will be kept, but will be secured through hashing and encryption. However, the contents varies from protocol to protocol.

**Onion**

* Card Number in Plaintext
* SHA-256 (SHA-3) hashed card number and pin number
* Public and private RSA keys (not pair) for Diffie Hellman key exchange
* Public Key for outer layer encryption
* Private Key for inner layer decryption
* Public Key for inner layer encryption
* The Onion
* Signature for the bank

**Funyun**

* ATM ID
* ATM Number of Bills
* The Funyun
* SHA-256 (3) hashed card number and pin number
* IV
* Dynamic AES Passkeys for decrypting Funyon

\*\*Keys for AES secure tunnels will be stored in RAM to keep away prying eyes.

Quick Reference code for database initialization:

"""Implements a Database interface for the bank server and admin interface"""

def \_\_init\_\_(self, db\_path="bank.json"):

self.path = db\_path

self.admin\_db = admin\_db.Admin\_DB(db\_path=db\_path)

def close(self):

"""close the database connection"""

pass

def init\_db(self):

"""initialize database with file at filepath"""

with open(self.path, 'w') as f:

f.write(json.dumps({'atms': {}, 'cards': {}, 'storage': {}}))

def exists(self):

if not self.exists():

self.init\_db()

def modify(self, table, k, subks, vs):

with open(self.path, 'r') as f:

db = json.loads(f.read())

try:

for subk, v in zip(subks, vs):

if k not in db[table]:

db[table][k] = {}

db[table][k][subk] = v

except KeyboardInterrupt:

return False

with open(self.path, 'w') as f:

f.write(json.dumps(db))

return True

def read(self, table, k, subk):

with open(self.path, 'r') as f:

db = json.loads(f.read())

try:

return db[table][k][subk]

except KeyError:

return None

**Communication**

All communications with the Bank will go over SWD to the BeagleBone Black (ATM), hence the import of the module ‘serial’ in most files within the protocol. Everything that will go to the ATM is secured by the AES tunnel originally secured by the Diffie Hellman key exchange.

**Section 6: The ATM**

**Communication**

Between the bank and the ATM, the most crucial information is sent, so therefore it must be encrypted and secured. To supplement this, the ATM has an AES key that encrypts and decrypts AES encryption between the ATM and the bank. If an attacker was listening in to communications, they could not decrypt the AES encryption between the bank and the ATM.

The alternate purpose of the ATM is to send requests and packages to and from the card. To secure this, communications are conducted over UART to the PSoC, but there are still attackers listening in on this side through card skimming (see ‘Attacks to Prevent’ in Challenge section 1). Therefore, the information that will have to be manually input by the user is just the 8-digit PIN number for verification.

**Interaction Process**

Ideally, the interaction for the Onion was thought up originally to be the most idealistically secure interaction. Originally, the idea was to:

* Card number and pin will be entered by the user.
* After 5 failed attempts to enter the pin number, the transaction is cancelled and the user is locked out of the account for 5 minutes (preventing brute forcing).
* Encrypt the card number, the pin, and salt together with SHA-256 hash to send to the bank with AES.
* The user may do what they must. Check balance, deposit, withdraw, change pin.
* Each session will have a user timeout so that if the transaction is not completed in 15 minutes it is ended. All information from the transaction will be deleted and the user will have to log back in. This is in the case that attackers target the individual ATM sessions for information. It will not be inactivity based, just a direct timeout.
* After transaction is complete, user information is deleted.

However, that is much easier said than done. In the implementation phase, interaction was mainly neglected due to lack of time, only a few features were able to be implemented. In hindsight, the 15 minute time out would have assisted in preventing side channel attacks temporarily for those whom did not access to the hardware, as well as locking out the user for 5 minutes to prevent brute forcing. The only things that were implemented are:

* Card number and PIN entered by user
* Encrypt the card number and PIN together with SHA-256 hash to send to the bank with AES.
* User may proceed with checking balance, withdrawing cash, and changing pin.
* After transaction complete, temporary user info is cleared, and keys for Dynamic AES are reset between bank and user.

**Section 7: The Card**

**A. Design**

The card is designed to continuously receive three letter commands from the atm until the card is taken out which functions as sort of a command prompt. All necessary environment variables are stored in flash in order to ensure security. Here are the predefined memory portions:

#define AESrow 150

#define ivrow 151

#define cardrow 152

#define passrow 153

Since were working with flash, we can only work in 128 bytes chunks of memory called “rows”(the row # acts as a index in memory for the beginning of each row when multiplied by 128), and we are storing our variables in the 150’s in order to avoid accidentally overwriting any key programming functions for the psoc that’s placed at the beginning.

**B.**  **For Loop Command Prompt**

As mentioned earlier the card has a continuously running for loop that receives three letter commands which are the following:

**“Prv”** - this command activates the provision protocol in the card and opens several buffers in order to store the sent variables.

**“Req”** - At the atms request I send over the 16 byte card number and the 32 bytes AES encrypted password hash..

**“Key”** - This command happens when the atm is sending over the balance I use for my next key. The current AES key and iv is loaded from memory and used to decrypt the balance. Then the balance is hashed together with the original AES key and is stored to memory as the key for the next transaction.

**C. Provisioning**

In the provisioning process we are receiving the following and simply writing them to their predefined memory addresses in flash.

-The production AES key

-The production iv

-The production card number

-The production pass

**D. Functions**

The following functions are used in order to aid in improving the efficiency of our program.

-**loadmem**(int row, int size, uint8\* buf): Given the row number, the size of the buffer, and the base pointer of the buffer it will load the first “size” number of bytes at the given row of memory into the passed buffer.

-**getvalidbytes**(uint8\_t\* buffer, int size): Given a base pointer to a buffer and a size, it will populate that buffer up to that size with external output. It will not proceed to fill the next address until it receives a valid input.

-**memcpy**: This function is not one of our written functions, but we use it many times throughout our program in order to pack all sent variables into one package.

For ex:

memcpy(bigboi, cardnum, 16);

memcpy(bigboi+16, cipherhp, 32);

Big boi is a 48 byte buffer, card 16, cipherhp 32. The 16 bytes of card number is copied into the first 16 bytes of bigboi buffer. Next the cipherhp buffer is copied into the big boi base pointer offset by 16 bytes so that we don’t overwrite the card number. This way we package all values into one buffer and don’t have to worry about sequentially sending items over serial and can just send one package.