

Cryptographic Analysis

Colin Rice

Samuel Milito

Jacob Shedd

December 5, 2012

1 Flaws and Exploit Enablers

There are a number of issues that, while not exploits in and of themselves, either allow exploits, or are issues within the system.

1.1 Bank Deposit Limits

Due to the incorrect way atm limits are implemented, completely on the server side and globally, the bank is limited to depositing \$1000 into any account. It cannot make multiple \$1000 deposits.

1.2 Poor Scaling

While not an issue with the current scale of the project, all account information is stored in a vector. With $O(n)$ lookups, this will become incredibly slow with very large amounts of users.

1.3 Overly Generous Timeouts

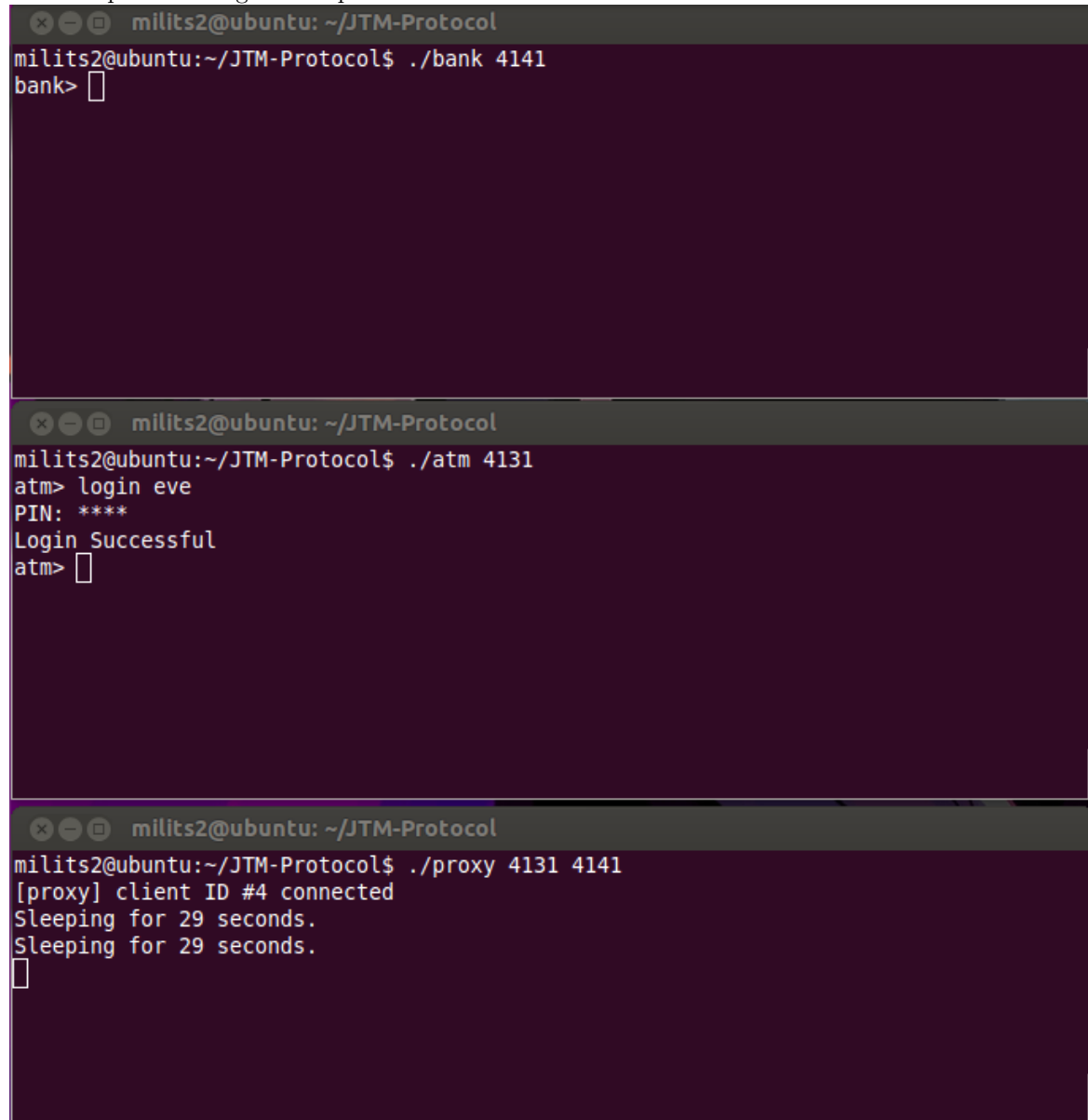
The timeout checking on both sides is overly generous. The ATM has a 30 second window

```
1 if(time(NULL) - messageTimeout < 30) // Bank Response needs to be in less than 30
   seconds.
2 {
3     // [...]
4 }
```

and the bank has no timeout handling at all. As such, packets can be held up to 30 seconds before being sent to the ATM, and indefinitely before being sent to the bank. This opens it up to much longer analysis than otherwise. While there is no specific exploit I have in mind, this is a much larger window than is needed for packets to be sent and received. In the proxy, a large amount of work can be done. Adding the following code in between receiving an ATM packet and forwarding the packet to the bank results in proper operation:

```
1 printf( "Sleeping for 29 seconds.\n" );
2 sleep( 29 );
```

An example of adding the sleeps is below.



The image displays three sequential terminal windows from a user named 'milits2' on an Ubuntu system, located in the directory '~/JTM-Protocol'.
The first window shows the command `./bank 4141` being executed, with the prompt changing to `bank>`.
The second window shows the command `./atm 4131` being executed. The prompt changes to `atm>`, followed by the command `login eve`. The output shows `PIN: ****` and `Login Successful`, with the prompt returning to `atm>`.
The third window shows the command `./proxy 4131 4141` being executed. The output shows `[proxy] client ID #4 connected`, followed by two lines of `Sleeping for 29 seconds.`, and then a blank line with the prompt.

1.4 Poor PIN Verification

PINs are supposed to be 6 characters, but any amount of zeros at the end are ignored. If a PIN is entered with less than the required characters, it pads it out with zeros:

```
1 | pin = getPin("PIN: ");
```

```

2| pin = pin.substr(0,6);
3| while(pin.length() < 6)
4| {
5|     pin = pin + "0";
6| }
7| pin = toNumbers(pin);

```

For example, a PIN of 678900 will accept 6789, 67890, or 678900.

1.5 Lack of Server Side Verification

On the server side, no verification is done that the received numbers are integers.

```

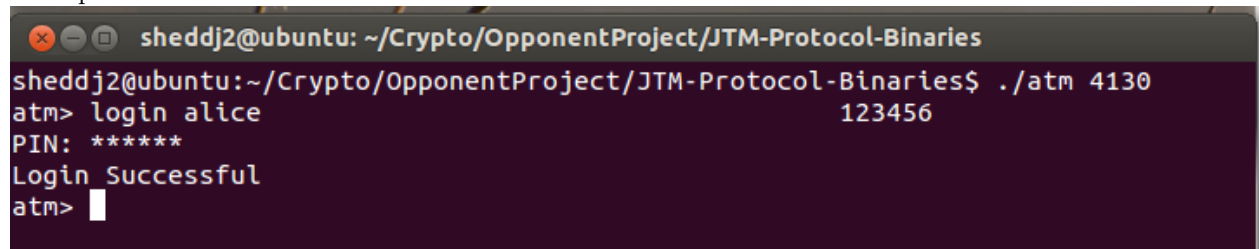
1| float b = (float) atof(info.at(4).c_str());

```

Bank balances should always be integers; there is no reason to accept floats.

1.6 Single Line Login

Example



A terminal window titled 'sheddj2@ubuntu: ~/Crypto/OpponentProject/JTM-Protocol-Binaries'. The user runs './atm 4130'. The ATM interface shows 'atm> login alice' followed by '123456' on the same line. Then 'PIN: *****' is displayed, followed by 'Login Successful' and 'atm>' with a cursor.

Because of the way the atm parses user input it is possible to have the user login and pin on the same line. This still allows the user to login but for anyone looking over their shoulder they would see the pin in cleartext.

1.7 Bank Deposits Floats

While the ATM only allows withdrawals of integer amounts, the bank allows for deposits of floats. There is no server side input verification other than making sure that it is a positive float. If there is a deposit at the bank with at least seven decimal places (e.g., 0.9999999), when the user checks their balance it will register as \$1 more than they previously had. They will then be able to withdraw the new amount, plus the dollar.

While this isn't very exploitable (the user would need to sit down at a bank's computer and deposit \$0.9999999 multiple times), it shows that the bank allows for malformed inputs.

In the following image, repeated deposits of \$0.9999999 were followed by depositing a whole amount. This rolled it over, and allowed the user to withdraw the whole dollar amount.

```
Terminal
sheddj2@ubuntu: ~/Crypto/OpponentProject/JTM-Protocol-Binaries

bank> deposit eve .99999
Deposit [ eve ]: $ .99999
bank> balance eve
Balance [ eve ]: $ 0.99999
bank> deposit eve .0000099999
Deposit [ eve ]: $ .0000099999
bank> balance eve
Balance [ eve ]: $ 1
bank> deposit eve .9999999
Deposit [ eve ]: $ .9999999
bank> balance eve
Balance [ eve ]: $ 1
bank> deposit eve 10
Deposit [ eve ]: $ 10
bank>

Logout Successful
atm> login eve
PIN: ****
Login Successful
atm> withdraw 2
Withdraw: $2
atm> balance
Balance: $0
atm> withdraw 1
Withdraw: $1
atm> withdraw 1
Error. Please contact the service team.
atm> balance
Balance: $1
atm> withdraw 1
Error. Please contact the service team.
atm> withdraw 10
Timeout: user inactivity has caused a timeout, the current user has now been logged out.
Logout Successful
atm> login eve
PIN: *****
Login Successful
atm> withdraw 10
Withdraw: $10
atm> balance
Balance: $1
atm> withdraw 1
Withdraw: $1
atm>
```

1.8 Key Management

The binaries do not have hardcoded keys. Instead of being cooked into the binaries, they read the keys from a special folder. This has two consequences. First, it makes it far easier to get copies of the keys. Second, it allows for on-the-fly key manipulation—if an attacker has access to the directory controlling the keys, they can be changed at runtime.

1.9 No HMAC

There is no HMAC being used. Messages are hashed, but it is simple SHA512, which is not passworded. If the encryption is broken, modifying a message is trivial, as the attacker can just recompute the hash.

1.10 Unhashed Card Contents

The card contents are never hashed. There is a `getCardHash` function that does no hashing:

```
1 std::string getCardHash(std::string cardFile)
2 {
3     std::ifstream card(cardFile.c_str());
4     std::string tempHash((std::istreambuf_iterator<char>(card), std::
5         istreambuf_iterator<char>()));
6
7     std::string cardHash = tempHash;
8     cardHash = cardHash.substr(0,128);
9     std::transform(cardHash.begin(), cardHash.end(), cardHash.begin(), ::toupper);
10    cardHash = toHex(cardHash);
11
12    return cardHash;
13 }
```

It just reads the first 128 characters from the file (or attempts to, there is no assurance that it has 128 characters) then converts it to uppercase, then lastly to hex. Note that the `toHex` function does no verification on the string; if it's fed invalid hex it just replaces all characters not 0-9, A-F with 0.

```
1 std::string toHex(std::string inputStr)
2 {
3     std::string retStr = "";
4     for(int i = 0; i < inputStr.length(); i++)
5     {
6         if(('0' <= inputStr[i] && inputStr[i] <= '9') || ('A' <= inputStr[i] &&
7             inputStr[i] <= 'F'))
8         {
9             retStr += inputStr[i];
10        }
11        else
12        {
13            retStr += "0";
14        }
15    }
16 }
```

```

16     return retStr;
17 }

```

1.11 RNG Flaws

Their random number generator when asked for 32 bytes of data outputs 32 random hex bytes. This means that each byte only has 16 (2^4) possibilities instead of 256 (2^8).

The end result is that everything generated using these functions (AES keys, nonces, etc.) only have 128 bits of randomness instead of the expected 256 bits of randomness.

```

1  std::string getRandom(int length)
2  {
3      std::string retStr = "";
4      CryptoPP::AutoSeededRandomPool rng;
5      int random;
6      int num = 0;
7      bool hex = true;
8
9      for(unsigned int i = 0; i < length; ++i)
10     {
11         random = (int) rng.GenerateByte();
12         if(hex)
13         {
14             // Generate Random Hex String
15             num = (int) (random % 16);
16             if(num < 10)
17             {
18                 retStr += (num + '0');
19             }
20             else
21             {
22                 retStr += ((num - 10) + 'A');
23             }
24         }
25         else
26         {
27             // Generate Random String With ASCII Range (48, 126)
28             num = (int) (random % (126 - 48));
29             retStr += (num + '0');
30         }
31     }
32
33     return retStr;
34 }

```

2 Exploits

2.1 Invalid Transfers

A transfer to an invalid user succeeds initially, as the checks are done out of order, namely the highlighted lines:

```

1  bool processTransfer (std::vector<std::string> info)

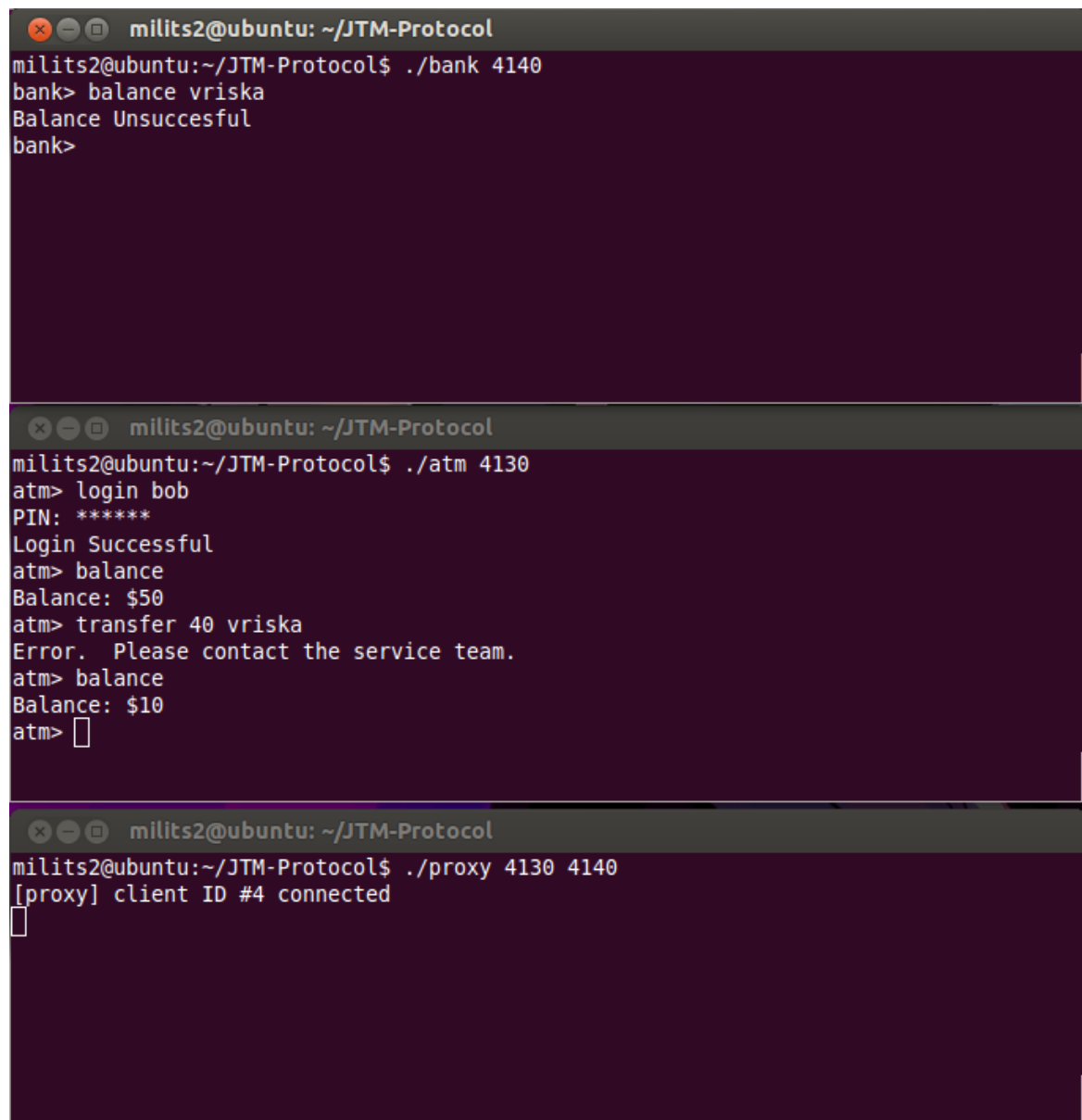
```

```

2| {
3|     float b = (float) atof(info.at(4).c_str());
4|     if(b > 1000.00) {
5|         return false;
6|     }
7|
8|     std::vector<Account>::iterator it;
9|     for (it = Database.begin(); it != Database.end(); it++) {
10|         if(it->get_un() == info.at(1) && it->get_logged_in() && b <= it->get_balance()
11|            && it->get_transfer() + b <= 1000.00) {
12|             it->reduce_balance(b);
13|             it->increase_transfer (b);
14|             std::vector<Account>::iterator foo;
15|             for (foo = Database.begin(); foo != Database.end(); foo++) {
16|                 if(foo->get_un() == info.at(5) && foo->get_balance() + b <= MAXBAL) {
17|                     foo->increase_balance (b);
18|                     return true;
19|                 }
20|             }
21|         }
22|     }
23|     return false;
24| }

```

The active user has their account debited before the targetted user has their account credited. The result is that issuing a transfer to an invalid user will result in the active ATM user losing money, which then vanishes entirely from the system. In the image, Bob has \$40 vanish when he attempts to transfer it to Vriska, a nonexistent user.



```
milits2@ubuntu: ~/JTM-Protocol
milits2@ubuntu:~/JTM-Protocol$ ./bank 4140
bank> balance vriska
Balance Unsuccessful
bank>

milits2@ubuntu:~/JTM-Protocol$ ./atm 4130
atm> login bob
PIN: *****
Login Successful
atm> balance
Balance: $50
atm> transfer 40 vriska
Error. Please contact the service team.
atm> balance
Balance: $10
atm>

milits2@ubuntu:~/JTM-Protocol$ ./proxy 4130 4140
[proxy] client ID #4 connected
```

2.2 Account Enumeration Via Transfer

\$0 transfers can be used to verify whether or not a given user has an account at the bank. For example, Alice tries to send \$0 to Eve (which succeeds) and Vriska (which fails), showing that Eve has an account, but Vriska does not. In the image, Alice verifies that Bob is a valid user (as the transfer succeeds) and that Vriska is not a valid user, as the transfer fails.


```
milits2@ubuntu: ~/JTM-Protocol
milits2@ubuntu:~/JTM-Protocol$ ./bank 4140
bank>

milits2@ubuntu: ~/JTM-Protocol
milits2@ubuntu:~/JTM-Protocol$ ./atm 4130
atm> login alice
PIN: *****
Login Successful
atm> transfer 0 bob
Transfer Successful
atm> transfer 0 vriska
Error. Please contact the service team.
atm>

milits2@ubuntu: ~/JTM-Protocol
milits2@ubuntu:~/JTM-Protocol$ ./proxy 4130 4140
[proxy] client ID #4 connected
```

2.3 Dysfunctional Card Verification

The actual contents of a card file are ignored. The only verification done on the card is if it has anything in it. That is, any card put into the machine will satisfy it, as long as it doesn't have a blank "magnetic strip" (i.e., contents). This is a massive security flaw, as it is a failure of two-factor security. 128 characters are read from the card file, and summarily ignored (after being sent as

encrypted plaintext). The login code is as follows:

```
1 bool login (std::vector<std::string> info)
2 {
3     std::vector<Account>::iterator it;
4     int pin = atoi(info.at(3).c_str());
5
6     for (it = Database.begin(); it != Database.end(); it++) {
7         if(it->get_un() == info.at(1) && it->get_pin() == pin && !it->get_logged_in
8             () && !it->get_locked()) {
9             it->set_logged_in_true ();
10            return true;
11        }
12        else if(it->get_un() == info.at(1) && it->get_pin() != pin && !it->get_logged_in
13            ()) {
14            it->increase_login_attempts();
15            if(it->get_login_attempts() >= 3) {
16                it->lock();
17            }
18            return false;
19        }
20    }
```

It verifies that the PIN matches, but never checks if the card actually matches with anything. In fact, the card is never used anywhere. The only verification on the card is that the file exists and isn't empty:

```
1 if(cardFile)
2 {
3     sendPacket = true;
4
5     username = temp;
6
7     cardHash = getCardHash(cardFilename);
8
9     pin = getPin("PIN: ");
10    pin = pin.substr(0,6);
11    while(pin.length() < 6)
12    {
13        pin = pin + "0";
14    }
15    pin = toNumbers(pin);
16 }
17 else
18 {
19     temp = "";
20     cout << "ATM card not found.\n";
21 }
```

This verifies that the card exists, and that is all. The card is eventually rolled into a packet (see below) then never used.

2.4 Login Info

All login info is sent in cleartext inside the encrypted packet. The most significant issue is that the PIN is sent unencrypted. The first 128 characters in the card file are sent, regardless of if the card file contains 128 characters or not. Note that the card contents are ignored.

None of the login information is hashed, so being able to decrypt messages reveals the full amount of information needed to compromise somebody's account. As it also sends account name, an attacker has account name, PIN, and the (unused) card number.

```
1 void* formATMPacket(char packet[], std::string command, std::string username, std::
    string cardHash, std::string pin, std::string item1, std::string item2, std::
    string atmNonce, std::string bankNonce)
2 {
3     std::vector<std::string> tempVector;
4     tempVector.push_back(command);
5     tempVector.push_back(username);
6     tempVector.push_back(cardHash);
7     tempVector.push_back(pin);
8     tempVector.push_back(item1);
9     tempVector.push_back(item2);
10    tempVector.push_back(atmNonce);
11    tempVector.push_back(bankNonce);
12    formPacket(packet, 1023, tempVector);
13 }
```

The packet sent on login and all other requests contains the unhashed username, card/account number, and PIN.

2.5 Crashing the Bank

Adding a single line of code to the proxy will crash the bank. The bank cannot handle a malformed handshake message. The follow section of code in the proxy:

```
1 while(1)
2 {
3     //read the packet from the ATM
4     // [...]
5
6     // Crashes bank
7     strcpy( packet, "" );
8
9     //forward packet to bank
10    // [...]
11
12    //get response packet from bank
13    // [...]
14
15    // Crashes ATM
16    strcpy( packet, "" );
17
18    //forward packet to ATM
19    // [...]
20 }
```

can be easily manipulated to crash the bank (or, similarly, the ATMs, as detailed slightly later). Adding the line of code before forwarding the packet to the bank crashes the bank as soon as an

ATM user attempts to log in.

```
create mode 100644 PermaLockoutViaCtrlC.png
milits2@ubuntu:~/Crypto-Final-Projects$ 
milits2@ubuntu: ~/JTM-Protocol
milits2@ubuntu:~/JTM-Protocol$ ./bank 4140
bank> terminate called after throwing an instance of 'CryptoPP::InvalidArgument'
  what(): RSA/OAEP-MGF1(SHA-1): ciphertext length of 0 doesn't match the required le
ngth of 768 for this key
Aborted (core dumped)
milits2@ubuntu:~/JTM-Protocol$ 
milits2@ubuntu: ~/JTM-Protocol
milits2@ubuntu:~/JTM-Protocol$ ./atm 4130
atm> login eve
PIN: ****
milits2@ubuntu:~/JTM-Protocol$ 
milits2@ubuntu: ~/JTM-Protocol
milits2@ubuntu:~/JTM-Protocol$ ./proxy 4130 4140
[proxy] client ID #4 connected
[proxy] fail to read packet length
[proxy] client ID #4 disconnected

```

Similarly, ATMs can be crashed by adding the line of code before forwarding the packet to the ATM.

```
milits2@ubuntu: ~/JTM-Protocol
milits2@ubuntu:~/JTM-Protocol$ ./bank 4140
bank>

milits2@ubuntu: ~/JTM-Protocol
milits2@ubuntu:~/JTM-Protocol$ ./atm 4130
atm> login eve
PIN: ****
terminate called after throwing an instance of 'CryptoPP::InvalidArgument'
  what():  RSA/OAEP-MGF1(SHA-1): ciphertext length of 0 doesn't match the required length of 768 for this key
Aborted (core dumped)
milits2@ubuntu:~/JTM-Protocol$

milits2@ubuntu: ~/JTM-Protocol
milits2@ubuntu:~/JTM-Protocol$ ./proxy 4130 4140
[proxy] client ID #4 connected
[proxy] client ID #4 disconnected
```

If both lines are left in, the bank crashes with a core dump, and the ATM crashes with no error message. This issue is due to not checking the validity of initial handshake messages. When it receives a blank message, it still tries to proceed with setting up a secure connection. The cryptographic functions crash, and the ATM and/or bank crash with them.

2.6 Race Conditions

The code as-is provides no protections on money, i.e., there are no mutex locks. For example, the withdrawl function:

```
1 bool processWithdraw (std::vector<std::string> info)
2 {
3     float b = (float) atof(info.at(4).c_str());
4     if(b > 1000.00) {
5         return false;
6     }
7
8     std::vector<Account>::iterator it;
9     for (it = Database.begin(); it != Database.end(); it++) {
10 ==> if(it->get_un() == info.at(1) && it->get_logged_in() && b <= it->get_balance()
    && it->get_withdraw() + b <= 1000.00) {
11         it->reduce_balance(b);
12         it->increase_withdraw (b);
13         return true;
14     }
15 }
16 return false;
17 }
```

That is, a user logged on multiple times could theoretically withdraw double the amount in their account, assuming that the timing went through correctly. They would need to pass the marked line at the same time on two consoles, and it will authorize withdrawal on two ATMs, regardless if it would go over the account balance or the daily \$1000 limit. The only protection against this is not allowing users to log in on multiple machines at a time. However, the login function is also susceptible to a race condition:

```
1 bool login (std::vector<std::string> info)
2 {
3     std::vector<Account>::iterator it;
4     int pin = atoi(info.at(3).c_str());
5
6     for (it = Database.begin(); it != Database.end(); it++) {
7 ==> if(it->get_un() == info.at(1) && it->get_pin() == pin && !it->get_logged_in()
    && !it->get_locked()) {
8         it->set_logged_in_true ();
9         return true;
10    }
11    else if(it->get_un() == info.at(1) && it->get_pin() != pin && !it->get_logged_in
    ()) {
12        it->increase_login_attempts();
13        if(it->get_login_attempts() >= 3) {
14            it->lock();
15        }
16        return false;
17    }
18 }
19 return false;
20 }
```

If the timing works out correctly, a user will pass all required checks (the highlighted line), and then `set_logged_in_true` will be called. During this time, any other ATMs can then theoretically pass the same checks. The end result is that a single user would be logged on at multiple ATMs. Due to

how fast all of the operations are (the data sets are miniscule), a demonstration was not feasible. This issue is exacerbated by the thirty second timeout window for packets (i.e., the bank accepts a packet if it's no more than 30 seconds old), as the proxy could hold packets, making sure that they are sent to the bank at identical times.

With a user logged on to multiple ATMs, they can abuse the race conditions to withdraw or transfer double their account balance.

2.7 Handshake Hijacking

When the bank and ATM establish a handshake, the ATM sends the bank a nonce encrypted with the bank's RSA key. The bank then sends back an encryption key and the nonce, as well as its own generated nonce. There is no explicit authentication.

Assuming an attacker has the ATM's public key, it is possible to masquerade as the bank. The attacker can simply send back a key encrypted with the ATM's public key and the nonce.

As there is no RSA (or other) authentication, if an attacker can guess the nonce, they can masquerade as the bank. Normally this would take 2^{256} attempts, but due to the aforementioned RNG flaw, it only takes 2^{128} attempts. Even worse, if the ATM is being run on an embedded system, entropy will be low upon startup, making it possible to guess the output of the system's random number generator.

If a successful masquerade is established, there are two crippling exploits, with some less important ones. The first is full account information obtained, as the proxy can then decrypt the message, which contains full cleartext login information for the user (see Login Info). The second is verifying all requests, including withdrawals for any amount. That is, masquerading as the bank compromises login information for all users who connect and also allows attackers to drain all of the money from any ATM.