John Quinn
CSE 40622
Final Project Report

Project Overview:

Target Functionality and Goals:

The aim of the this project was to create an over-the-network bank account management system. Normal users can log on, view their account info, transfer money between accounts, and view and update personal information. Administrative users can add and remove funds from accounts, create new accounts and users, and view everyone's account information. Cryptographic techniques are used in order to protect sensitive information. Someone listening on the network should not be able to get personal information, account numbers, passwords, or any other sensitive data.

Cryptographic Computations:

There are several cryptographic techniques that are used in this program. When users start the program, they are prompted for a login name and password. The server has an RSA public/private key pair. Each of these keys is stored in a file. Ideally, there would be some sort of trusted digital signature authority that would sign the public key as it is transmitted to the client. Since I do not have access to such a service, I am going to assume that the client already has the server's public key in a file. The client then uses this RSA public key to encrypt the username and password. When the server receives this information, it decrypts it using its private key. Then the server can authenticate the user by checking the information in the database. This way, the client and server can be sure they are communicating with each other. This step is important because it gives some protection against a man in the middle attack on the Diffie-Hellman key agreement scheme. Because the client encrypts its secrets with the known public key of the server, only the server can decrypt them. Likewise, each user should be the only one's who know their own login information.

The Diffie-Hellman key agreement protocol is used to establish a shared secret between the client and the server. This shared secret is meant to last for only the duration of the session. A new public/private key pair is generated by both the client and the server every time there is a new login. Because RSA was used to authenticate, there is some protection against an active attacker.

After the shared secret key is established, the client and server can send commands to each other. This uses AES 256 bit symmetric key encryption in the cipher-block chaining mode. This mode works well because messages that are sent are always a fixed length. After each message is encrypted and stored in a buffer, a HMAC is computed on the encrypted data for integrity. This HMAC is then added to the buffer. This HMAC is computed using EVP_sha256() algorithm that is available in OpenSSL. The key for the HMAC is generated by the server and transmitted to the client using AES once the shared secret has been established. The IV values used in the AES CBC algorithm are also added to the buffer in the clear so that the messages can be decrypted. These methods ensure both confidentiality and integrity.

Message Format:

Messages sent from the client to the server are 256 bytes in length. The first byte is always a code that corresponds the the command. Bytes 2-159 can contain any data that might need to be sent to the server, which is specific to the command. Bytes 160-191 contain the 32 byte HMAC. Bytes 192-200 are padding. Bytes 200-231 contain the IV values that are sent in the clear.

Messages sent from the server to the client are always 1024 bytes in length. The first 928 bytes can be data that is being sent back as a response. The next 32 bytes are an HMAC of the encrypted data. The next 32 bytes are the IV values used in encryption. The rest of the bytes serve as padding.

Goals of Project:

The goals of the project were to make this scheme as secure as possible. I wanted to preserve both secrecy and integrity of the messages. I also wanted to protect against man in the middle attacks and allow for users to establish session keys.

Project Outcome:

Project Setup:

The project is programmed in c/c++ and uses the OpenSSL library for cryptographic operations. A sqlite3 database is used to store data on the server side. The program was tested on the student CSE machines, mainly student03.cse.nd.edu. When testing the program, both the client and server application were run on the same machine in different ssh sessions.

There are 3 executables produced by the makefile. Two of them are the client and server applications. The last is an application called keygen that generates a private/public RSA key pair and writes each key to a file. This can be used to regenerate the keys for the first part of the program as needed. The sqlite database is called accounts.db. There are two scripts, build.sql and clean.sql, which can be used to build and clean the database respectively.

Results of Implementation In Respect to Project Goals:

In regards to performance, transfer of data across the server seems to be very responsive once the shared secret keys are established. There is a bit of overhead establishing the shared secret, but it is worth it once everything is initialized. Security-wise, both secrecy and integrity of the messages is obtained. AES is a strong encryption scheme, and the cipher-block chaining mode is non-deterministic. New random IV values are generated every time a message is sent. HMAC works well for integrity, as each party has access to a secret key. Using Diffie-Hellman to agree on a secret is a good method in this case. An attempt is made to protect against man in the middle attacks by using RSA and an account password system. It would be better to have a trusted certificate authority sign the public key, but for our purposes we assume the client already has the key. As far as scalability goes, only one user can access the server at a time. Focus was placed on making sure the cryptographic protocols functioned as intended.

Difficulties Encountered:

Getting OpenSSL to work as intended was no easy task, especially with all of the components used. I had to figure out how to get the AES, HMAC, Diffie-Hellman, and RSA functions to all work the way they should.

As a result, some of the checking of inputs isn't as good as it should be due to time constraints. I thought it would be more beneficial to get the cryptographic components working first. The checking of inputs for the admin mode is especially lacking. I'm assuming administrators would be more likely to know what they are doing.

Source Code:

packet_structs.h

```
struct user_pass
{
	char name[64];
	char pass[64];
};

struct transfer
{
	char sender[32];
	char receiver[32];
	int amount;
	char date[32];
};

struct all_transfers
{
	int num_transfers;
	struct transfer transfer_list[8];
};

struct user_info
{
	char uname[64];
	char first[64];
	char last[64];
```

```
        char email[64];
        char telephone[64];
        int isAdmin;
};

struct accounts
{
        int type;
        int funds;
        char accountno[64];
};

struct all_accounts
{
        int num_accounts;
        struct accounts account_list[8];
};
```

crypto_lib.h

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

//OpenSSL for Encryption
#include <openssl/aes.h>
#include <openssl/rand.h>
#include <openssl/hmac.h>
#include <openssl/evp.h>
#include <openssl/rsa.h>
#include <openssl/dh.h>
#include <openssl/bn.h>
#include <openssl/rsa.h>
#include <openssl/pem.h>

void gen_secret_key(unsigned char * key)
{
        // get a 256 bit key
        RAND_bytes(key, 32);
}

void gen_iv(unsigned char * iv)
{
        // get a 256 bit iv - needs to be same length as key
        // generated by sender
        RAND_bytes(iv, 32);
}
```

```
void aes_cbc_sec_enc(unsigned char * input, unsigned char * output, int length, unsigned char * key,
unsigned char * iv)
{
        // CBC encryption
        AES_KEY theKey;
        AES_set_encrypt_key(key, 256, &theKey);
        AES_cbc_encrypt(input, output, length, &theKey, iv, AES_ENCRYPT);
}

void aes_cbc_sec_dec(unsigned char * input, unsigned char * output, int length, unsigned char * key,
unsigned char * iv)
{
        // CBC decryption
        AES_KEY theKey;
        AES_set_decrypt_key(key, 256, &theKey);
        AES_cbc_encrypt(input, output, length, &theKey, iv, AES_DECRYPT);
}

void compute_hash(unsigned char * input, int input_len, unsigned char * output, unsigned char *
userKey)
{
        unsigned int out_len;
        HMAC(EVP_sha256(), userKey, 32, input, input_len, output, &out_len);
}

int verify_hash(unsigned char * data, int data_len, unsigned char * hash, unsigned char *userKey)
{
        unsigned char output[32];
        unsigned int out_len;
        HMAC(EVP_sha256(), userKey, 32, data, data_len, output, &out_len);

        if (memcmp(hash, output, 32)==0)
        {
                printf("HMAC verified.\n");
                return 1;
        }

        return 0;
}
```

keygen.cpp

```
// Program that generates an RSA public key/private key pair and saves them to files.
#include <stdlib.h>
#include <stdio.h>
#include <openssl/rsa.h>
#include <openssl/engine.h>
#include <openssl/pem.h>

int main (void)
```

```
{
	// generates public and private keys and saves them to a file
	RSA *theKey = RSA_generate_key(512, 3, NULL, NULL);

	printf("KEY SIZE: %d\n", RSA_size(theKey));

	FILE * privateKeyOut =fopen("privkey.pem", "w");
	FILE * publicKeyOut = fopen("pubkey.pem", "w");

	PEM_write_RSAPrivateKey(privateKeyOut, theKey, NULL, NULL, 0, NULL, NULL);
	PEM_write_RSAPublicKey(publicKeyOut, theKey);

	fclose(privateKeyOut);
	fclose(publicKeyOut);

	publicKeyOut = fopen("pubkey.pem", "r");
	privateKeyOut =fopen("privkey.pem", "r");

	//test if key can be read back
	RSA *Read_test = RSA_new();
	PEM_read_RSAPrivateKey(privateKeyOut, &Read_test, NULL, NULL);
	PEM_read_RSAPublicKey(publicKeyOut, &Read_test, NULL, NULL);

	return 0;
}
```

client.cpp

```
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <err.h>

//OpenSSL for Encryption
#include <openssl/aes.h>
#include <openssl/rand.h>
#include <openssl/hmac.h>
#include <openssl/evp.h>
#include <openssl/rsa.h>
#include <openssl/dh.h>
#include <openssl/rsa.h>
#include <openssl/pem.h>
#include <openssl/err.h>
```

```
#include "packet_structs.h"
#include "crypto_lib.h"

void print_options()
{
  printf("Here is a list of available commands.\n\t h - list help menu\n\t a - view accounts
information\n\t p - view personal profile information\n\t t - transfer funds between two accounts\n\t i -
change personal profile information\n\t l - view transaction log\n\t q - quit application\n");
}

void print_options_admin()
{
  printf("Here is a list of available commands.\n\t h - list help menu\n\t v - view all accounts for a
user\n\t c - create a new account \n\t u - create a new user \n\t a - add funds to an account\n\t r - remove
funds from an account \n\t q - quit application\n");
}

void clear_send_buff(unsigned char * sendBuf)
{
  for (int i=0; i<256; i++)
  {
    sendBuf[i]=0x0;
  }
}

void print_user_info(struct user_info * current_user)
{
  printf("User: %s\n", (*current_user).uname);
  printf("Name: %s %s\n", (*current_user).first, (*current_user).last);
  printf("Email: %s\n", (*current_user).email);
  printf("Telephone: %s\n", (*current_user).telephone);
}

void print_accounts_info(struct all_accounts *user_accounts)
{
  int i;
  for (i=0; i<(*user_accounts).num_accounts; i++)
  {
    struct accounts current_account = (*user_accounts).account_list[i];
    printf("Account no: %s ", current_account.accountno);
    if (current_account.type == 0)
    {
      printf("Savings ");
    }
    else if (current_account.type == 1)
    {
      printf("Checking ");
    }
    printf("Balance: %d\n",current_account.funds);
  }
}

void print_transfer_info(struct all_transfers *transfers)
{
  int i;
  printf("Here are your most recent transfers (up to 8): \n");
  for (i=0; i<(*transfers).num_transfers; i++)
  {
    struct transfer current_transfer = (*transfers).transfer_list[i];
    printf("From: %s   To: %s   Amount: %d   Time: %s\n", current_transfer.sender,
current_transfer.receiver, current_transfer.amount, current_transfer.date);
  }
}

int do_request_reply(int sockfd, unsigned char * sendBuf, unsigned char * recBuf, unsigned char *
theKey, unsigned char * hashKey)
{
  unsigned char encSendBuf[256];
  unsigned char encRecBuf[1024];

  unsigned char iv[32];

  gen_iv(iv);

  memcpy(encSendBuf+200, iv, 32);

  unsigned char hashVal[32];

  aes_cbc_sec_enc(sendBuf, encSendBuf, 160, theKey, iv);

  compute_hash(encSendBuf, 160, hashVal, hashKey);
  memcpy(encSendBuf+160, hashVal, 32);

  if(send(sockfd, encSendBuf, 256, 0)==-1)
  {
    printf("An error occurred sending the server request info.");
    return -1;
  }

  if(recv(sockfd, encRecBuf, 1024, 0)==-1)
  {
    printf("Error occurred receiving the server response.");
    return -1;
  }

  memcpy(iv, encRecBuf+960, 32);
  memcpy(hashVal, encRecBuf+928, 32);
  if (verify_hash(encRecBuf, 928, hashVal, hashKey)==0)
  {
    printf("WARNING: HMAC not verified.  Data may be tampered with.\n");
  }

  aes_cbc_sec_dec(encRecBuf, recBuf, 928, theKey, iv);

  return 0;
}

int check_authentication(int sockfd)
{
    int authenticate=0;
    char name[64];
    char password[64];
    struct user_pass log_attempt;
    struct user_pass enc_log_attempt;

    //prompt user for password
    printf("Enter usename:");
    scanf("%s", name);
    printf("Enter password:");
    scanf("%s", password);

    //copy into struct
    strncpy(log_attempt.name, name, strlen(name));
    strncpy(log_attempt.pass, password, strlen(password));

    // NULL termintate strings
    log_attempt.name[strlen(name)]='\0';
    log_attempt.pass[strlen(password)]='\0';

    // get the RSA public key
    RSA *serverkey = RSA_new();
    FILE * publicKeyOut = fopen("pubkey.pem", "r");
    PEM_read_RSAPublicKey(publicKeyOut, &serverkey, NULL, NULL);
    fclose(publicKeyOut);

    if(RSA_public_encrypt(64, (unsigned char*)&log_attempt.name, (unsigned
char*)&enc_log_attempt.name, serverkey, RSA_NO_PADDING)==-1)
    {
      char err[64];
      ERR_load_crypto_strings();
      ERR_error_string(ERR_get_error(), err);
      fprintf(stderr, "Error encrypting message: %s\n", err);
    }
    if(RSA_public_encrypt(64, (unsigned char*)&log_attempt.pass, (unsigned
char*)&enc_log_attempt.pass, serverkey, RSA_NO_PADDING)==-1)
```

```
  {
   char err[64];
   ERR_load_crypto_strings();
   ERR_error_string(ERR_get_error(), err);
   fprintf(stderr, "Error encrypting message: %s\n", err);
  }

  // send packet of uname/pass
  if(send(sockfd, &enc_log_attempt, sizeof(struct user_pass), 0)==-1)
  {
   printf("An error occurred sending the login info.");
   return 0;
  }

  // check authentication
  if(recv(sockfd, &authenticate, sizeof(int), 0)==-1)
  {
   printf("Error occurred with authentication response.");
   return 0;
  }
  //give user response
  if(authenticate==0)
  {
   printf("Username/password combination not recognized.  Please try again. \n");
  }
  else
  {
   printf("Authentication succeeded.  Welcome %s.\n", log_attempt.name);
  }

  return authenticate;
}

int main(void)
{
 int sockfd = 0;
 struct sockaddr_in serv_addr;

 if((sockfd = socket(AF_INET, SOCK_STREAM, 0))< 0)
 {
  printf("\n Error : Could not create socket \n");
  return 1;
 }

 serv_addr.sin_family = AF_INET;
 serv_addr.sin_port = htons(5000);
 serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
 if(connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr))<0)
 {
  printf("\n Error : Connect Failed \n");
  return 1;
 }

 int authenticate = 0;
 int isAdmin;

 // get Public RSA key from server
 authenticate=check_authentication(sockfd);
 if (authenticate==0)
 {
  return 0;
 }

 // check for admin privileges
 if(recv(sockfd, &isAdmin, sizeof(int), 0)==-1)
 {
  printf("Error occurred with admin info response.");
  return 0;
 }

 // after we authenticated, loop to accept commands
 if (authenticate==1)
 {
  char input[8];
  fgets(input, 8, stdin);
  unsigned char sendBuf[256];
  unsigned char recBuf[1024];

  unsigned char theKey[32];
  unsigned char hashKey[32];

  printf("Generating shared secret...\n");

  // set up shared secret key using Diffie Hellman
  // setup p and g
  if(recv(sockfd, recBuf, 1024, 0)==-1)
  {
   printf("Error occurred with authentication response.");
   return 0;
  }

  unsigned char serverPublicKey[128];
  unsigned char P[128];
  unsigned char G[1];
  unsigned char publicKey[128];

  memcpy(P, recBuf, 128);
  memcpy(serverPublicKey, recBuf+128, 128);
  memcpy(G, recBuf+256, 1);

  DH *privkey=DH_new();

  privkey->p = BN_new();
  privkey->g = BN_new();

  BN_bin2bn(G, 1, privkey->g);
  BN_bin2bn(P, 128, privkey->p);

  if(DH_generate_key(privkey))
  {
   BN_bn2bin(privkey->pub_key, publicKey);
  }

  memcpy(sendBuf, publicKey, 128);

  if(send(sockfd, sendBuf, 256, 0)==-1)
  {
   printf("An error occurred sending the request to the client.");
   return -1;
  }

  BIGNUM * pubKeyBN = BN_new();
  BN_bin2bn(serverPublicKey, 128, pubKeyBN);

  unsigned char sharedKey[128];
  DH_compute_key(sharedKey, pubKeyBN, privkey);

  memcpy(theKey, sharedKey, 32);

  unsigned char encRecBuf[1024];
  if(recv(sockfd, encRecBuf, 1024, 0)==-1)
  {
   printf("Error occurred receiving the server response.");
   return -1;
  }

  unsigned char iv[32];
  memcpy(iv, encRecBuf+960, 32);
  aes_cbc_sec_dec(encRecBuf, recBuf, 928, theKey, iv);

  memcpy(hashKey, recBuf, 32);

  printf("Enter 'h' to list options.");

  if (isAdmin==0)
  {
   while(1)
```

```
      {
       clear_send_buff(sendBuf);
       printf("\nPlease enter a command:");
       fgets(input, 8, stdin);
       if (input[0]=='\n')
       {
         continue;
       }
       if (input[0]=='h')
       {
         // help menu
         print_options();
       }
       else if (input[0]=='a')
       {
         // view accounts of user
         sendBuf[0]='a';

         if (do_request_reply(sockfd, sendBuf, recBuf, theKey, hashKey)==-1) return 0;

         struct all_accounts * user_accounts;
         user_accounts = (struct all_accounts *) recBuf;
         printf("NUMBER OF ACCOUNTS %d\n", (*user_accounts).num_accounts);
         print_accounts_info(user_accounts);

       }
       else if (input[0]=='p')
       {
         // lets user view their personal information
         sendBuf[0]='p';

         if (do_request_reply(sockfd, sendBuf, recBuf, theKey, hashKey)==-1) return 0;

         struct user_info * current_user;
         current_user = (struct user_info *) recBuf;
         print_user_info(current_user);

       }
       else if (input[0]=='t')
       {
         // lets user transfer funds between their own accounts
         sendBuf[0]='t';

         struct transfer current_transfer;
         printf("Account Transer:\nYou must own the sending account.\nPlease enter account no. of
sending account:\n");
         fgets(current_transfer.sender, 30, stdin);
         if (current_transfer.sender[strlen(current_transfer.sender)-1]=='\n')
current_transfer.sender[strlen(current_transfer.sender)-1]='\0';
```

```
          printf("Please enter account no. of receiving account: \n");
          fgets(current_transfer.receiver, 30, stdin);
          if (current_transfer.receiver[strlen(current_transfer.receiver)-1]=='\n')
current_transfer.receiver[strlen(current_transfer.receiver)-1]='\0';
          fflush(stdin);
          printf("Please enter the amount you would like to transfer: \n");
          scanf("%d", &current_transfer.amount);
          fflush(stdin);

          memcpy(sendBuf+1, &current_transfer, sizeof(struct transfer));
          if (do_request_reply(sockfd, sendBuf, recBuf, theKey, hashKey)==-1) return 0;

          char result[128];
          memcpy(result, recBuf, 128);
          printf("%s\n", result);
        }
        else if (input[0]=='i')
        {
          // lets user change their personal information
              sendBuf[0]='i';
          char change[8];
          printf("Enter a choice:\n\t e - change email \n\t t - change telephone\n\t q - cancel request\n:");
          fgets(change, 8, stdin);
          // do choice
          char new_entry[64];
          if (change[0]=='e')
          {
            sendBuf[1]='e';
            printf("Please enter a new email: \n");
            fgets(new_entry, 64, stdin);
            if (new_entry[strlen(new_entry)-1]=='\n') new_entry[strlen(new_entry)-1]='\0';
            memcpy(sendBuf+2, new_entry, 64);
            if (do_request_reply(sockfd, sendBuf, recBuf, theKey, hashKey)==-1) return 0;
            else printf("Email changed.\n");
          }
          else if (change[0]=='t')
          {
            sendBuf[1]='t';
            printf("Please enter a new phone number: \n");
            fgets(new_entry, 64, stdin);
            if (new_entry[strlen(new_entry)-1]=='\n') new_entry[strlen(new_entry)-1]='\0';
            memcpy(sendBuf+2, new_entry, 64);
            if (do_request_reply(sockfd, sendBuf, recBuf, theKey, hashKey)==-1) return 0;
            else printf("Telephone changed.\n");
          }
          else if (change[0]=='q')
          {
            continue;
          }
```

```
      else
      {
        printf("Error: invalid choice.");
        continue;
      }
    }
    else if (input[0]=='l')
    {
      // lets user view a log of their past transactions
      sendBuf[0]='l';

      if (do_request_reply(sockfd, sendBuf, recBuf, theKey, hashKey)==-1) return 0;

      struct all_transfers * transfers;
      transfers = (struct all_transfers *) recBuf;
      print_transfer_info(transfers);
    }
    else if (input[0]=='q')
    {
      // exits the applications safely, closing connection with server
      sendBuf[0]='q';
      if (do_request_reply(sockfd, sendBuf, recBuf, theKey, hashKey)==-1) return 0;
      else printf("Goodbye.\n");
      close(sockfd);
      break;
    }
    else
    {
      printf("Error: command not recoginzed.");
    }

  }
}
else if (isAdmin==1)
{
  while(1)
  {
    clear_send_buff(sendBuf);
    printf("\nPlease enter a command:");
    fgets(input, 8, stdin);
    if (input[0]=='\n')
    {
      continue;
    }
    if (input[0]=='h')
    {
      // help menu
      print_options_admin();
    }
```

```
else if (input[0]=='v')
{
  // view all all accounts for a user
  sendBuf[0]='v';
  char username[64];

  printf("Enter a username: ");
  scanf("%s", username);

  memcpy(sendBuf+8, username, 64);
  if (do_request_reply(sockfd, sendBuf, recBuf, theKey, hashKey)==-1) return 0;

  struct all_accounts * user_accounts;
  user_accounts = (struct all_accounts *) recBuf;
  if ((*user_accounts).num_accounts==0)
  {
    printf("No accounts exist for the specified user.\n");
  }
  else
  {
    printf("NUMBER OF ACCOUNTS %d\n", (*user_accounts).num_accounts);
    print_accounts_info(user_accounts);
  }

}
else if (input[0]=='c')
{
  // create a new account
  sendBuf[0]='c';

  char accountno[64];
  char username[64];
  char type[4];

  printf("Enter username of account owner: ");
  scanf("%s", username);
  printf("Enter an account no: ");
  scanf("%s", accountno);
  printf("Enter 0 for savings, 1 for checking: ");
  scanf("%s", &type);

  memcpy(sendBuf+8, username, 64);
  memcpy(sendBuf+72, accountno, 64);
  memcpy(sendBuf+136, type, 4);
  if (do_request_reply(sockfd, sendBuf, recBuf, theKey, hashKey)==-1) return 0;
  printf("Account successfully created. \n");

}
else if (input[0]=='u')
```

```
{
 // create a new user
 sendBuf[0]='u';
 char username[32];
 char first[32];
 char last[32];
 char password1[32];
 char password2[32];

 printf("Enter new username: ");
 scanf("%s", username);
 printf("Enter a first and last name: ");
 scanf("%s %s", first, last );
 printf("Enter password: ");
 scanf("%s", password1);
 printf("Enter password: ");
 scanf("%s", password2);
 if (strncmp(password1,password2,32)!=0)
 {
  printf("Error, passwords do not match.  Account not created.\n");
 }
 else
 {
  memcpy(sendBuf+8, username, 32);
  memcpy(sendBuf+40, password1, 32);
  memcpy(sendBuf+72, first, 32);
  memcpy(sendBuf+104, last, 32);
  if (do_request_reply(sockfd, sendBuf, recBuf, theKey, hashKey)==-1) return 0;
  printf("Account successfully created. \n");
 }

}
else if (input[0]=='a')
{
 // add funds to an account
 sendBuf[0]='a';
 char accountno[64];
 char funds[10];

 printf("Enter account no.: \n");
 scanf("%s", accountno);
 printf("Enter amount: \n");
 scanf("%s", funds);

 memcpy(sendBuf+8, accountno, 64);
 memcpy(sendBuf+72, funds, 10);

 if (do_request_reply(sockfd, sendBuf, recBuf, theKey, hashKey)==-1) return 0;
 printf("%s\n", (char *) recBuf);
```

```
        }
        else if (input[0]=='r')
        {
          // remove funds from an account
          sendBuf[0]='r';
          char accountno[64];
          char funds[10];

          printf("Enter account no.: \n");
          scanf("%s", accountno);
          printf("Enter amount: \n");
          scanf("%s", funds);

          memcpy(sendBuf+8, accountno, 64);
          memcpy(sendBuf+72, &funds, 10);

          if (do_request_reply(sockfd, sendBuf, recBuf, theKey, hashKey)==-1) return 0;
          printf("%s\n", (char *) recBuf);
        }
        else if (input[0]=='q')
        {
          // exits the applications safely, closing connection with server
          sendBuf[0]='q';
          if (do_request_reply(sockfd, sendBuf, recBuf, theKey, hashKey)==-1) return 0;
          else printf("Goodbye.\n");
          close(sockfd);
          break;
        }
        else
        {
          printf("Error: command not recoginzed.");
        }

      }
    }

  }


  return 0;
}
```

server.cpp

```
include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
```

```
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sqlite3.h>

//OpenSSL for Encryption
#include <openssl/aes.h>
#include <openssl/rand.h>
#include <openssl/hmac.h>
#include <openssl/evp.h>
#include <openssl/rsa.h>
#include <openssl/dh.h>
#include <openssl/bn.h>
#include <openssl/rsa.h>
#include <openssl/pem.h>
#include <openssl/err.h>

#include "packet_structs.h"
#include "crypto_lib.h"

//variable for sql queries
//made global so they can be used by all functions easily
char query[1024];
int query_len=1024;
sqlite3_stmt *pStatement;
const char * pzTail;

int check_authentication(int connfd, sqlite3 *bank_db, struct user_pass enc_log_attempt, struct
user_info * current_user)
{
  int authenticate=0;
  int isAdmin=0;

  // get the RSA keypair
  RSA *serverkey = RSA_new();
  FILE * publicKeyOut = fopen("pubkey.pem", "r");
  FILE * privateKeyOut =fopen("privkey.pem", "r");
  PEM_read_RSAPublicKey(publicKeyOut, &serverkey, NULL, NULL);
  PEM_read_RSAPrivateKey(privateKeyOut, &serverkey, NULL, NULL);
  fclose(publicKeyOut);
  fclose(privateKeyOut);

  // decrypt data from client into another user_pass struct
  struct user_pass log_attempt;

  if(RSA_private_decrypt(64, (unsigned char*)&enc_log_attempt.name, (unsigned
char*)&log_attempt.name, serverkey, RSA_NO_PADDING)==-1)
```

```
  {
    char err[64];
    ERR_load_crypto_strings();
    ERR_error_string(ERR_get_error(), err);
    fprintf(stderr, "Error encrypting message: %s\n", err);
  }
  if(RSA_private_decrypt(64, (unsigned char*)&enc_log_attempt.pass, (unsigned
char*)&log_attempt.pass, serverkey, RSA_NO_PADDING)==-1)
  {
    char err[64];
    ERR_load_crypto_strings();
    ERR_error_string(ERR_get_error(), err);
    fprintf(stderr, "Error encrypting message: %s\n", err);
  }

  sprintf(query, "select * from users where uname='%s'", log_attempt.name);
  sqlite3_prepare(bank_db, query, query_len, &pStatement, &pzTail);
  while (sqlite3_step(pStatement)!=SQLITE_DONE)
  {
   char *password;
   password = (char *)sqlite3_column_text(pStatement,1);
   if (strcmp(password, log_attempt.pass)==0)
   {
    authenticate=1;
    strncpy((*current_user).uname, (char *) sqlite3_column_text(pStatement,0), 64);
    strncpy((*current_user).first, (char *) sqlite3_column_text(pStatement,2), 64);
    strncpy((*current_user).last, (char *) sqlite3_column_text(pStatement,3), 64);
    strncpy((*current_user).email, (char *) sqlite3_column_text(pStatement,4), 64);
    strncpy((*current_user).telephone, (char *) sqlite3_column_text(pStatement,5), 64);
    (*current_user).isAdmin = (int) sqlite3_column_int(pStatement,6);
   }
  }
  sqlite3_finalize(pStatement);

  if ((*current_user).isAdmin==1)
  {
   isAdmin=1;
  }
  else
  {
   isAdmin=0;
  }

  if(send(connfd, &authenticate, sizeof(int), 0)==-1)
  {
   printf("Error occured sending authentication info.");
   return 0;
  }
  if(send(connfd, &isAdmin, sizeof(int), 0)==-1)
  {
    printf("Error occured sending authentication info.");
    return 0;
  }

  return authenticate;
}

void get_user_accounts(sqlite3 *bank_db, struct user_info current_user, struct all_accounts *
user_accounts)
{
  sprintf(query, "select * from accounts where uname='%s'", current_user.uname);
  sqlite3_prepare(bank_db, query, query_len, &pStatement, &pzTail);
  int account_ct=0;
  while (sqlite3_step(pStatement)!=SQLITE_DONE)
  {
    struct accounts account;
    account.type=(int) sqlite3_column_int(pStatement,1);
    account.funds=(int) sqlite3_column_int(pStatement,2);
    strncpy(account.accountno, (char *) sqlite3_column_text(pStatement,3), 64);

    (*user_accounts).account_list[account_ct]=account;

    (*user_accounts).num_accounts=++account_ct;
  }
  sqlite3_finalize(pStatement);
}

void admin_get_user_accounts(sqlite3 *bank_db, char * user_name, struct all_accounts *
user_accounts)
{
  sprintf(query, "select * from accounts where uname='%s'", user_name);
  sqlite3_prepare(bank_db, query, query_len, &pStatement, &pzTail);
  int account_ct=0;
  (*user_accounts).num_accounts=0;
  while (sqlite3_step(pStatement)!=SQLITE_DONE)
  {
    struct accounts account;
    account.type=(int) sqlite3_column_int(pStatement,1);
    account.funds=(int) sqlite3_column_int(pStatement,2);
    strncpy(account.accountno, (char *) sqlite3_column_text(pStatement,3), 64);

    (*user_accounts).account_list[account_ct]=account;

    (*user_accounts).num_accounts=++account_ct;
  }
  sqlite3_finalize(pStatement);
}
```

```
void admin_add_user(sqlite3 * bank_db, char * username, char * password, char * first, char * last)
{
  char default_string[12]="UNKNOWN";
  sprintf(query, "insert into users values ('%s', '%s', '%s', '%s', '%s', '%s', 0 )", username, password, first,
last, default_string, default_string );
  sqlite3_prepare(bank_db, query, query_len, &pStatement, &pzTail);
  sqlite3_step(pStatement);
  sqlite3_finalize(pStatement);
}

void admin_create_new_account(sqlite3 *bank_db, char * username, char *accountno, char *type)
{
  int type_int=atoi(type);
  sprintf(query, "insert into accounts values ('%s', %d, 0, '%s')", username, type_int, accountno);
  sqlite3_prepare(bank_db, query, query_len, &pStatement, &pzTail);
  sqlite3_step(pStatement);
  sqlite3_finalize(pStatement);
}

void get_user_transfers(sqlite3 * bank_db, struct user_info current_user, struct all_transfers *
user_transfers)
{
  sprintf(query, "select * from transfer_log where uname='%s' order by time desc",
current_user.uname);
  sqlite3_prepare(bank_db, query, query_len, &pStatement, &pzTail);
  int trans_ct=0;
  while (sqlite3_step(pStatement)!=SQLITE_DONE && trans_ct<8)
   {
    struct transfer the_transfer;
    the_transfer.amount= (int) sqlite3_column_int(pStatement,3);
    strncpy(the_transfer.sender, (char *) sqlite3_column_text(pStatement,1),32);
    strncpy(the_transfer.receiver, (char *) sqlite3_column_text(pStatement,2) ,32);
    strncpy(the_transfer.date, (char *) sqlite3_column_text(pStatement,5), 32);

    (*user_transfers).transfer_list[trans_ct]=the_transfer;
    (*user_transfers).num_transfers=++trans_ct;
   }
  sqlite3_finalize(pStatement);
}

void attempt_transfer(sqlite3 * bank_db, struct transfer current_transfer, struct user_info current_user,
char *result)
{
  struct accounts sender;
  struct accounts receiver;

  // gather info for transfer
  sprintf(query, "select * from accounts where account_no='%s' and uname='%s'",
current_transfer.sender, current_user.uname);
```

```
  sqlite3_prepare(bank_db, query, query_len, &pStatement, &pzTail);
  int check=0;
  int rc;
  if(rc=sqlite3_step(pStatement)!=SQLITE_DONE)
  {
    fflush(stdout);
    check++;
    sender.type = (int) sqlite3_column_int(pStatement,1);
    sender.funds = (int) sqlite3_column_int(pStatement,2);
    strncpy(sender.accountno, current_transfer.sender, 64);
  }
  sqlite3_finalize(pStatement);
  if(check==0)
  {
    sprintf(result, "Error, sender does not have rights to account with given number.\n");
    return;
  }

  sprintf(query, "select * from accounts where account_no='%s'", current_transfer.receiver);
  sqlite3_prepare(bank_db, query, query_len, &pStatement, &pzTail);
  check=0;
  if(sqlite3_step(pStatement)!=SQLITE_DONE)
  {
    check++;
    receiver.type = (int) sqlite3_column_int(pStatement,1);
    receiver.funds = (int) sqlite3_column_int(pStatement,2);
    strncpy(receiver.accountno, current_transfer.receiver, 64);
  }
  sqlite3_finalize(pStatement);
  if(check==0)
  {
    sprintf(result, "Error, receiving account does not exits.\n");
    return;
  }

  // check for sufficient funds
  if (current_transfer.amount > sender.funds)
  {
    sprintf(result, "Error, insufficient funds.\n");
    return;
  }

  // update accounts, log transfer
  sender.funds-=current_transfer.amount;
  receiver.funds+=current_transfer.amount;

  sprintf(query, "update accounts set funds='%d' where account_no='%s'", sender.funds,
sender.accountno);
  sqlite3_prepare(bank_db, query, query_len, &pStatement, &pzTail);
  sqlite3_step(pStatement);
  sqlite3_finalize(pStatement);

  sprintf(query, "update accounts set funds='%d' where account_no='%s'", receiver.funds,
receiver.accountno);
  sqlite3_prepare(bank_db, query, query_len, &pStatement, &pzTail);
  sqlite3_step(pStatement);
  sqlite3_finalize(pStatement);

  sprintf(query, "insert into transfer_log (send_from, send_to, amount, uname) values('%s', '%s', %d,
'%s')", sender.accountno, receiver.accountno, current_transfer.amount, current_user.uname);
  sqlite3_prepare(bank_db, query, query_len, &pStatement, &pzTail);
  sqlite3_step(pStatement);
  sqlite3_finalize(pStatement);

  sprintf(result, "Transfer was a success.\n");
  return;

}

int admin_add_funds(sqlite3 *bank_db, char * account_no, char * funds)
{
  struct accounts currentAccount;
  int funds_int = atoi(funds);

  sprintf(query, "select * from accounts where account_no='%s'", account_no);
  sqlite3_prepare(bank_db, query, query_len, &pStatement, &pzTail);
  int check=0;
  if(sqlite3_step(pStatement)!=SQLITE_DONE)
  {
    check++;
    currentAccount.funds = (int) sqlite3_column_int(pStatement,2);
  }
  sqlite3_finalize(pStatement);
  if(check==0)
  {
    return 0;
  }

  currentAccount.funds+= funds_int;

  sprintf(query, "update accounts set funds=%d where account_no='%s'", currentAccount.funds ,
account_no);
  sqlite3_prepare(bank_db, query, query_len, &pStatement, &pzTail);
  sqlite3_step(pStatement);
  sqlite3_finalize(pStatement);

  return 1;
```

```
}

int admin_remove_funds(sqlite3 *bank_db, char * account_no, char * funds)
{
  struct accounts currentAccount;
  int funds_int = atoi(funds);

  sprintf(query, "select * from accounts where account_no='%s'", account_no);
  sqlite3_prepare(bank_db, query, query_len, &pStatement, &pzTail);
  int check=0;
  if(sqlite3_step(pStatement)!=SQLITE_DONE)
  {
    check++;
    currentAccount.funds = (int) sqlite3_column_int(pStatement,2);
  }
  sqlite3_finalize(pStatement);
  if(check==0)
  {
    return 0;
  }

  currentAccount.funds-= funds_int;

  sprintf(query, "update accounts set funds=%d where account_no='%s'", currentAccount.funds ,
account_no);
  sqlite3_prepare(bank_db, query, query_len, &pStatement, &pzTail);
  sqlite3_step(pStatement);
  sqlite3_finalize(pStatement);

  return 1;

}

void update_phone(sqlite3 *bank_db, struct user_info * currentUser, char * updateBuf)
{
  strncpy((*currentUser).telephone, updateBuf, 30);
  sprintf(query, "update users set phone='%s' where uname='%s'", (*currentUser).telephone,
(*currentUser).uname);
  sqlite3_prepare(bank_db, query, query_len, &pStatement, &pzTail);
  sqlite3_step(pStatement);
  sqlite3_finalize(pStatement);
}

void update_email(sqlite3 *bank_db, struct user_info * currentUser, char * updateBuf)
{
  strncpy((*currentUser).email, updateBuf, 30);
  sprintf(query, "update users set email='%s' where uname='%s'", (*currentUser).email,
(*currentUser).uname);
  sqlite3_prepare(bank_db, query, query_len, &pStatement, &pzTail);
  sqlite3_step(pStatement);
  sqlite3_finalize(pStatement);
}

int encrypt_and_send(int connfd, unsigned char * sendBuf, unsigned char * theKey, unsigned char *
hashKey)
{
  unsigned char encSendBuf[1024];
  unsigned char iv[32];

  gen_iv(iv);

  memcpy(encSendBuf+960, iv, 32);

  unsigned char hashVal[32];

  aes_cbc_sec_enc(sendBuf, encSendBuf, 928, theKey, iv);

  compute_hash(encSendBuf, 928, hashVal, hashKey);
  memcpy(encSendBuf+928, hashVal, 32);

  if(send(connfd, encSendBuf, 1024, 0)==-1)
  {
    printf("An error occurred sending the request to the client.");
    close(connfd);
    return -1;
  }
  return 1;
}

int main(void)
{
  sqlite3 *bank_db;

  int rc;
  rc = sqlite3_open("accounts.db", &bank_db);
  if (rc)
  {
    fprintf(stderr, "Failed to open banking database: %s\n", sqlite3_errmsg(bank_db));
    sqlite3_close(bank_db);
    return 0;
  }

  int listenfd = 0,connfd = 0;
  struct sockaddr_in serv_addr;
  int numrv;
  listenfd = socket(AF_INET, SOCK_STREAM, 0);

  printf("socket retrieve success\n");
```

```
memset(&serv_addr, '0', sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(5000);
bind(listenfd, (struct sockaddr*)&serv_addr,sizeof(serv_addr));

if(listen(listenfd, 10) == -1){
    printf("could not establish ability to listen\n");
    return 0;
}

//generate RSA keypair


while(1)
{
  connfd = accept(listenfd, (struct sockaddr*)NULL ,NULL); // accept awaiting request
  fprintf(stdout, "Welcome to the bank.\n");

  while(1)
  {
    struct user_pass log_attempt;
    if(recv(connfd, &log_attempt, sizeof(struct user_pass), 0)==-1)
    {
      printf("An error occurred receiving the password.");
      close(connfd);
      break;
    }
    else
    {
        // check authentication from database
        int authenticate = 0;
        struct user_info current_user;

        authenticate=check_authentication(connfd, bank_db, log_attempt, &current_user);

        if (authenticate==1)
        {
          // vars for keys
          unsigned char theKey[32];
          gen_secret_key(theKey);
          unsigned char hashKey[32];
          gen_secret_key(hashKey);

          // buffers for sending and receiving
          unsigned char sendBuf[1024];
          unsigned char recBuf[256];
          unsigned char encRecBuf[256];
```

```
// set up shared secret key using Diffie Hellman

unsigned char publicKey[128];
unsigned char P[256];

gen_secret_key(P);

unsigned long G = 2;

DH* dh = DH_new();
dh->p = BN_new();
dh->g = BN_new();

BN_set_word(dh->g, G);
BN_bin2bn(P, 128, dh->p);
if(DH_generate_key(dh))
{
  BN_bn2bin(dh->pub_key, publicKey);
}

unsigned char P_send[128];
unsigned char G_send[1];

BN_bn2bin(dh->p, P_send);
BN_bn2bin(dh->g, G_send);

if(memcmp(P_send, P, 128)==0) printf("BLOCKS MATCH P.\n");
else printf("Blocks do not match P.\n");

if(memcmp(G_send, &G, 1)==0) printf("BLOCKS MATCH G.\n");
else printf("Blocks do not match G.\n");

memcpy(sendBuf, P_send, 128);
memcpy(sendBuf+128, publicKey, 128);
memcpy(sendBuf+256, G_send, 1);

// send public DH info to client
if(send(connfd, sendBuf, 1024, 0)==-1)
{
 printf("An error occurred sending the request to the client.");
 close(connfd);
 break;
}

// get the client's public key back
if(recv(connfd, recBuf, 256, 0)==-1)
{
   printf("Error occurred receiving the server response.");
```

```
    return -1;
}

unsigned char clientPublicKey[128];
memcpy(clientPublicKey, recBuf, 128);
BIGNUM * pubKeyBN = BN_new();
BN_bin2bn(clientPublicKey, 128, pubKeyBN);

unsigned char sharedKey[128];
DH_compute_key(sharedKey, pubKeyBN, dh);

memcpy(theKey, sharedKey, 32);

memcpy(sendBuf, hashKey, 32);

unsigned char iv[32];
unsigned char encSendBuf[1024];
gen_iv(iv);
memcpy(encSendBuf+960, iv, 32);
aes_cbc_sec_enc(sendBuf, encSendBuf, 928, theKey, iv);

if(send(connfd, encSendBuf, 1024, 0)==-1)
{
  printf("An error occurred sending the request to the client.");
  close(connfd);
  break;
}

while(1)
{
    if(recv(connfd, encRecBuf, 256, 0)==-1)
    {
      printf("An error occurred receiving the request from the client.\n");
      close(connfd);
      break;
    }
    else{

      // decrypt message
      unsigned char iv[32];
      memcpy(iv, encRecBuf+200, 32);

      unsigned char hashVal[32];
      memcpy(hashVal, encRecBuf+160, 32);

      if (verify_hash(encRecBuf, 160, hashVal, hashKey)==0)
      {
        printf("WARNING: HMAC not verified.  Data may be tampered with.\n");
      }
```

```
aes_cbc_sec_dec(encRecBuf, recBuf, 160, theKey, iv);

char command = (char) recBuf[0];

if (current_user.isAdmin==0)
{
 if (command=='a')
 {
  // get user account information
  struct all_accounts user_accounts;
  get_user_accounts(bank_db, current_user, &user_accounts);

  memcpy(sendBuf, &user_accounts, sizeof(struct all_accounts));
  if (encrypt_and_send(connfd, sendBuf, theKey,hashKey)==-1) return 0;
 }
 else if (command=='p')
 {
  // get user personal info
  memcpy(sendBuf, &current_user, sizeof(struct user_info));
  if (encrypt_and_send(connfd, sendBuf, theKey, hashKey)==-1) return 0;
 }
 else if (command=='t')
 {
  //attempt to do an account transfer
  struct transfer current_transfer;
  memcpy(&current_transfer, recBuf+1, sizeof(struct transfer));

  char result[128];
  attempt_transfer(bank_db, current_transfer, current_user, result);

  memcpy(sendBuf, result, 128);
  if (encrypt_and_send(connfd, sendBuf, theKey, hashKey)==-1) return 0;
 }
 else if (command=='i')
 {
  char choice = recBuf[1];
  char updateBuf[30];
  memcpy(updateBuf, recBuf+2, 30);
  if (choice=='e')
  {
   update_email(bank_db, &current_user, updateBuf);
   memcpy(sendBuf, "success", 20);
   if (encrypt_and_send(connfd, sendBuf, theKey, hashKey)==-1) return 0;
  }
  else if (choice=='t')
  {
   update_phone(bank_db, &current_user, updateBuf);
   memcpy(sendBuf, "success", 20);
```

```
               if (encrypt_and_send(connfd, sendBuf, theKey, hashKey)==-1) return 0;
              }
             }
             else if (command=='l')
             {
              struct all_transfers user_transfers;
              get_user_transfers(bank_db, current_user, &user_transfers);

              memcpy(sendBuf, &user_transfers, sizeof(struct all_transfers));
              if (encrypt_and_send(connfd, sendBuf, theKey,hashKey)==-1) return 0;
             }
             else if (command=='q')
             {
              memcpy(sendBuf, "success", 20);
              if (encrypt_and_send(connfd, sendBuf, theKey, hashKey)==-1) return 0;
              close(connfd);
              break;
             }
            }
            else if (current_user.isAdmin==1)
            {
             int result=0;
             if (command=='v')
             {
              struct all_accounts user_accounts;
              admin_get_user_accounts(bank_db, (char *)recBuf+8, &user_accounts);

              memcpy(sendBuf, &user_accounts, sizeof(struct all_accounts));
              if (encrypt_and_send(connfd, sendBuf, theKey,hashKey)==-1) return 0;
             }
             else if (command=='c')
             {
              admin_create_new_account(bank_db, (char *)recBuf+8, (char *)recBuf+72, (char
*)recBuf+136);
              memcpy(sendBuf, "success", 20);
              if (encrypt_and_send(connfd, sendBuf, theKey, hashKey)==-1) return 0;
             }
             else if (command=='u')
             {
              admin_add_user(bank_db, (char *) recBuf+8, (char *) recBuf+40, (char *) recBuf+72,
(char *) recBuf+104);

              memcpy(sendBuf, "success", 20);
              if (encrypt_and_send(connfd, sendBuf, theKey, hashKey)==-1) return 0;
             }
             else if (command=='a')
             {
              result = admin_add_funds(bank_db, (char *) recBuf+8, (char *)recBuf+72);
              if (result==1)
```

```
                  {
                    memcpy(sendBuf, "Funds added.", 32);
                  }
                  else
                  {
                    memcpy(sendBuf, "Account doesn't exist.", 32);
                  }
                  if (encrypt_and_send(connfd, sendBuf, theKey, hashKey)==-1) return 0;
                }
                else if (command=='r')
                {
                  result = admin_remove_funds(bank_db, (char *) recBuf+8, (char *)recBuf+72);
                  if (result==1)
                  {
                    memcpy(sendBuf, "Funds removed.", 32);
                  }
                  else
                  {
                    memcpy(sendBuf, "Account doesn't exist.", 32);
                  }
                  if (encrypt_and_send(connfd, sendBuf, theKey, hashKey)==-1) return 0;
                }
                else if (command=='q')
                {
                  memcpy(sendBuf, "success", 20);
                  if (encrypt_and_send(connfd, sendBuf, theKey, hashKey)==-1) return 0;
                  close(connfd);
                  break;
                }
              }

            }
          }
        }
        authenticate=0;
      }
    }
    sleep(1);
  }

  close(connfd);
  close(listenfd);
  sqlite3_close(bank_db);

  return 0;
}
```

Description of Data:

The program was tested on the accounts.db sqlite database. The script build.sql was used to build the database and populate it with sample data. The functionality of the program was tested manually. All functions seem to work as intended.