Christopher Barbosa

Introduction/Project description:
The goal of the chat project was to put what we had learned so far on cryptographic protocols such as:

1. Diffie-Hellman
2. Digital Signatures
3. Message Authentication
4. SSH, etc.

in order to develop a working chat program in c which provides security to both parties utilizing the protocols aforementioned. We were tasked with implementing authentication and integrity between parties(MACs/HMAC), message secrecy through encyrption and decryption(using AES, etc.) in particular the following was explicitly asked of us:

```
I've given you a skeleton which does very basic chat stuff: Depending on the invocation,
it will listen for connections, or make one with another host. Beyond that, it just sends
and receives text,displaying each message in a log window. It will be up to you to:

- Write some sort of handshake protocol to setup ephemeral keys
(your protocol should have perfect forward secrecy!).
- Mutual authentication, using public key cryptography.
- After authentication, each message should be encrypted and tagged with a message
authentication code. You may also want to take measures to prevent replay attacks.


I think SSH will be a good model on which to base your protocol. In particular,
don't use PKI (public-key infrastructure,with certificates and such), and instead assume
that communicating parties have already exchanged public keys. However, implementing
deniable authentication would be a nice touch (and is something SSH does not provide).
If you want to use 3DH, you can find an example in dh-example.c.
```

Therefore, the remaining of this report will be dedicated to showing how I implemented the security protocols(with code included), and my thought process going into each step.

1 (Identity authentication)**.**

Solution.
Firstly we want to authenticate both parties are who they say they are. This can be implemented quite easily using some sort of key exchange, in this case, I chose to use 3-Diffie-Hellman(3DH). 3DH, is an extension of the standard DH and relies on the exchange of 3 separate keys in order to provide stronger security as opposed to DH. The protocol 3DH involves the following steps:

1. **Generation of long-term and Ephemeral keys:** each party generates a long term key-pair (public/private) and an ephemeral key pair for each given session. The long-term keys are used for authentication purposes, while the ephemeral keys are used to ensure the existence of forward secrecy.
2. **Exchange of each public key:** both long-term and ephemeral public keys are sent to the other party

3. **Computation of shared secret:** both parties compute a shared secret based on the keys of the other, to guarantee authentication.

4. **Derivation of session key:** The shared secret is then used to derive a session key, which was then used for encryption and decryption of the messages.

IMPLEMENTATION:
In the code itself, I defined a listener flag that was used to indicated which of the two thread I wanted to do what(1 for client, else server). In client I executed the following(symmetric in server):

```c
printf("client connection initiated...\n");
      NEWZ(sk_c); //client private key
      NEWZ(Ckey);
      dhGen(sk_c, Ckey);
      //for testing
      //printf("Ckey: ");
      //gmp_printf("%d\n", Ckey);
      printf("send CKey\n");
      sendKeyOverSocket(sockfd, 1, Ckey);


      //store the recieved key
      mpz_t Skey;
      mpz_init(Skey);
      printf("receive Skey\n");
      receiveKeyOverSocket(sockfd, Skey);
      //gmp_printf("%d\n", Skey); for testing


      //client ephemeral key
      NEWZ(pk_c);
      NEWZ(CephKey);
      dhGen(pk_c, CephKey);
      //for testing
      //printf("CephKey: ");
      //gmp_printf("%d\n", CephKey);
      printf("send CephKey\n");
      sendKeyOverSocket(sockfd, 2, CephKey);


      mpz_t SephKey;
      mpz_init(SephKey);
      printf("receive SephKey\n");
      receiveKeyOverSocket(sockfd, SephKey);
      //gmp_printf("%d\n", SephKey); for testing

      //now 3dh stuff
      unsigned char keybuf[32];
      size_t buflen = 32;

      dh3Final(sk_c, Ckey, pk_c, CephKey, Skey, SephKey, keybuf, buflen);

      SS = malloc(buflen);
      memcpy(SS, keybuf, buflen);
```

Where: first I generate the long term key for client, send it to the server, receive the long-term key from the server, generate the client ephemeral key, send it to server, receive the servers ephemeral key, and finally compute the 3DH final session key and shared secret(SS). below is the code which I implemented in order to send the keys and receive the keys:

```c
static void sendKeyOverSocket(int sockfd, int message_type, mpz_t key){

  int message_type_net = htonl(message_type);
  if(send(sockfd, &message_type_net, sizeof(message_type_net), 0) < 0){
    perror("failed␣to␣send␣message␣type");
    return;
  }

  char* key_str = mpz_get_str(NULL, 10, key);
  int key_len = strlen(key_str);
  if((send(sockfd, &key_len, sizeof(int), 0)) < 0){
    error("failed␣to␣send␣key␣length");
  }

  if((send(sockfd, key_str, key_len, 0)) < 0){
    error("failed␣to␣send␣key␣data");
  }
  free(key_str);
}
```

the above works by tagging the message using the variable message_type. The tag is used to let the reading thread know which value is associated to what; for example without the tags, say the first thread(client) sends both values over the network before the second thread(server) has a chance to read the first one, then there is no way of know what value is the long term key or ephemeral key and this will cause issues.

```c
static void receiveKeyOverSocket(int sockfd, mpz_t key){
  int message_type_net;

  if(recv(sockfd, &message_type_net, sizeof(message_type_net), 0) <= 0){
    perror("failed␣to␣receive␣message␣type");
    return;
  }

  int key_len;
  if(recv(sockfd, &key_len, sizeof(int), 0) <= 0){
    perror("failed␣to␣recieve␣key␣length");
    return;
  }

  char* key_str = (char*)malloc(key_len + 1);
  if(recv(sockfd, key_str, key_len, 0) <= 0){
    perror("failed␣to␣receive␣key␣data");
    free(key_str);
    return;
  }
  key_str[key_len] = '\0';

  if(mpz_set_str(key, key_str, 10) != 0){
    fprintf(stderr, "failed␣to␣convert␣key␣string␣to␣mpz_t");
    free(key_str);
    return;
  }

  free(key_str);
}
```

The above has to first read the message type of the received data and act from there, converting the value into a mpz_t type etc.
It is important to note that I do not explicitly make sure that both the client and server compute the same value for SS(checked later), which may be an issue. The logic behind this was that I defined SS as the following:

```
static unsigned char* SS = NULL;
```

so I thought that for whichever thread that puts a value for this first, due to it being static, this value could not be changed within main thereafter, or else it will cause an issue. I assumed that it worked because the only way this program worked was when I was able to see that both values were computed to be the same in client and server. However, my logic here could be false. ☐

2 (message encryption and decryption).

Solution.
Sort of arbitrarily I chose to use AES-256(in CBC-mode) for encryption and decryption.
IMPLEMENTATION
The logic and code for encryption and decryption is pretty self explanatory so not much to explain here:

```
  int encryption(unsigned char* pt, int pt_len, unsigned char* key, unsigned
      char* IV, unsigned char* ct){
  EVP_CIPHER_CTX *ctx;
  int len, ct_len;

  printf("encrypting message\n");
  if(!(ctx = EVP_CIPHER_CTX_new())){
    error("ciphertext creation error");
  }

  if(IV == NULL){
    if(RAND_bytes(IV, EVP_MAX_IV_LENGTH) != 1){
      error("error generating IV");
    }
  }


  if(EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, IV) != 1){
    error("aes encryption error");
  }
  else if(EVP_EncryptUpdate(ctx, ct, &len, pt, pt_len) != 1){
    error("encryption update error");
  }
  else{
    ct_len = len;
  }

  if(EVP_EncryptFinal_ex(ctx, ct + len, &len) != 1){
    error("encryption error");
  }
  else{
    ct_len += len;
  }

  EVP_CIPHER_CTX_free(ctx);
  return ct_len;
}
```

```
    int decryption(unsigned char* ct, int ct_len, unsigned char* key, unsigned
        char* IV, unsigned char* pt){
  EVP_CIPHER_CTX *ctx;
  int len, pt_len;
  //for testing
  printf("decrypting message\n");

  if(!(ctx = EVP_CIPHER_CTX_new())){
    error("ciphertext creation error");
  }
  else if(EVP_DecryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, IV) != 1){
    error("aes decryption error");
  }
  else if(EVP_DecryptUpdate(ctx, pt, &len, ct, ct_len) != 1){
    error("decryption update error");
  }
  else{
    pt_len = len;
  }
  if(EVP_DecryptFinal_ex(ctx, pt + len, &len) != 1){
    ERR_print_errors_fp(stderr);
    error("decryption error");
  }
  else{
    pt_len += len;
  }

  EVP_CIPHER_CTX_free(ctx);

  return pt_len;
}
```

Theoretically the above should use the provided cryptographic protocol in order to correctly encrypt and decrypt the messages sent by the client and server. Important thing to note is that: 1. I generate a new initialization vector with each session, which ensures proper and secure encryption, 2. I don't explicitly return the ciphertext or the plaintext from the functions and instead return the lengths of these things(could be an issue), this subtlety in decision will be made more clear when we discuss the message authentication and Integrity in the next section. □

3 (MAC and message integrity).

Solution.

For this project I used HMAC-SHA256, which combines the hash generated by SHA-256 with the incorporation of a key(SS) for authentication. In case of this project, it was critical to implement some sort of MAC, as the messages could be susceptible to man in the middle(MITM) attacks/redirects, etc. Therefore, we use the MAC so that we can ensure that the parties that are communicating are the ones that computed the SS earlier.

IMPLEMENTATION

firstly I needed to generate the HMAC in the first place using the following:

```
    //compute the MAC for the messages
unsigned char* computeMAC(unsigned char* data, int data_len, unsigned char*
   key, unsigned char* mac_out){
  unsigned int mac_len = SHA256_DIGEST_LENGTH;

  if(HMAC(EVP_sha256(), key, 32, data, data_len, mac_out, &mac_len) == NULL){
    error("HMAC␣calculation␣error");
  }

  return mac_out;
}
```

From here we need to make sure to MAC both incoming and outgoing messages from/to client/server, so I modified the sendmessage and revcmessage functions as follows:

```
    size_t len = strlen(message);
    //ADDED FOR ENC/DEC stuff
    unsigned char ct[1024];
    unsigned char IV[EVP_MAX_IV_LENGTH];
    int ct_len = encryption((unsigned char*)message, len, SS, IV, ct);

    //compute MAC on ct so we do MAC(ENC)
    unsigned char mac[SHA256_DIGEST_LENGTH];
    computeMAC(ct, ct_len, SS, mac);

    unsigned char message_with_IV_and_MAC[ct_len + EVP_MAX_IV_LENGTH +
       SHA256_DIGEST_LENGTH];
    //copy each of the message, and MAC, etc correctly
    //using the following offsets
    memcpy(message_with_IV_and_MAC, IV, EVP_MAX_IV_LENGTH);
    memcpy(message_with_IV_and_MAC + EVP_MAX_IV_LENGTH, ct, ct_len);
    memcpy(message_with_IV_and_MAC + EVP_MAX_IV_LENGTH + ct_len, mac,
       SHA256_DIGEST_LENGTH);
```

The above is what I added to the sendmessage function. Most of the code is simply setting up the parameters in order to perform the MAC, but it is important to note that I take the convention of:

```
MAC(ENC(m))
```

which is crutial. Like mentioned in class, if we do not do the above an attacker can eventually figure out the MAC and subsequently decrypt the message. So as you can see in the last 3 lines, I carefully copy over the memory for each of the pieces; the IV, the ciphertext, and the MAC. Similarly, in the recvmessage function I added(most) the following:

```
    size_t maxlen = 1024;
    unsigned char ct[maxlen]; /* might add \n and \0 */
    ssize_t nbytes;
    unsigned char IV[EVP_MAX_IV_LENGTH];

    while (1) {
        if ((nbytes = recv(sockfd, ct, maxlen, 0)) == -1)
            error("recv␣failed");

        if (nbytes == 0) {
            /* XXX maybe show in a status message that the other
             * side has disconnected. */
            return 0;
        }

        memcpy(IV, ct, EVP_MAX_IV_LENGTH);

        int ct_len = nbytes - EVP_MAX_IV_LENGTH - SHA256_DIGEST_LENGTH;
        unsigned char* nct = ct + EVP_MAX_IV_LENGTH;


        unsigned char received_mac[SHA256_DIGEST_LENGTH];

        memcpy(received_mac, ct + ct_len + EVP_MAX_IV_LENGTH,
           SHA256_DIGEST_LENGTH);
        //for testing
        //printf("received MAC: ");
        //for(int i = 0; i < SHA256_DIGEST_LENGTH; i++){
          //printf("%02x", received_mac[i]);
        //}
        //printf("\n");

        //compute MAC of the received ct
        unsigned char mac[SHA256_DIGEST_LENGTH];
        computeMAC(nct, ct_len, SS, mac);
        //for testing
        //printf("computed MAC: ");
        //for(int i = 0; i < SHA256_DIGEST_LENGTH; i++){
          //printf("%02x", mac[i]);
        //}
        //printf("\n");

        //verify the mac
        //this was not working for some reason even though they were the same
        if(memcmp(mac, received_mac, SHA256_DIGEST_LENGTH) != 0){
          fprintf(stderr, "MAC␣verification␣failed(possible␣tampering");
          continue;//ignore the message and move on
        }

        //decrypt ct
        unsigned char pt[maxlen];
        //was nbytes - EVP_MAX_IV_LENGTH
            int pt_len = decryption(nct, ct_len, SS, IV, pt);
        //int pt_len = decryption(ct, nbytes, SS, IV, pt);
        pt[pt_len] = 0;

        char* msg = malloc(pt_len +1);
        memcpy(msg, pt, pt_len +1);
```

The important part of the above is that it parsely the input recieved and extracts the ct, the MAC, etc. and using that to compute it's own MAC. If we find that the two values are the same then there was no funny business from a third party and so we allow the chatting to continue, else we output a message to let the parties know. □

4 (Conclusion).

Solution.
in the following conclusion I want to address how well the implemented chat program was able to:

1. Implement perfect forward secrecy
2. Implement mutual authenticaion
3. Proper usage of MACs

I feel that my implentation does indeed provide perfect forward secrecy(1) becuase with using 3DH I compute new ephemeral and long-term keys each session so there is no way of using previous session keys to decyrpt future messages. Next, I do think mutual authentication is achieved(2) as the shared key is derived using the keys received from the other party. From my own testing, the only way the above works is when the SS is computed to be the same value by both parties or else the program does not run but it is possible that I might be overlooking an obscure case. Lastly, I feel like I was able to provide proper implementation of message authentication codes(3) as for each message I used HMAC-SHA256 for secure encryption with a fresh IV each message, and I followed the encrypt then MAC paradigm correctly. In general, I think my implementation was robust in the sense that it achieved everything it was supposed to, however there are some issues that were mentioned in the above sections in the way I chose to implement things, and so it could have been improved on if I had a more in-depth understanding of the material. □