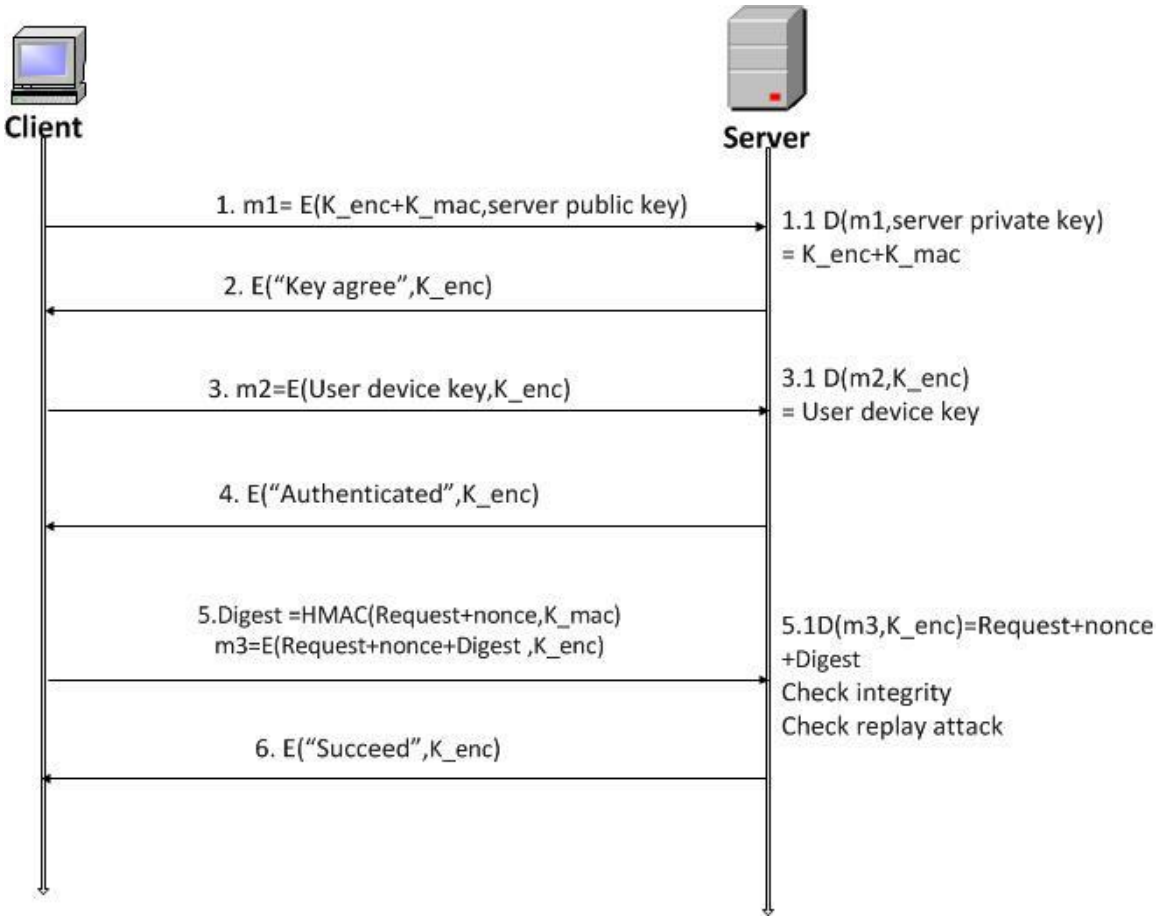
Student Name: Huangxin Wang

G#: 00773152

ISA656 Project2 Milestone#2

1. Implementation of protocols

The implementation of each step in the designed protocol will be show in details in the following sections. The protocol designed in milestone #1 is shown below.



1.1 Step1: Client negotiate key with server.

The key to be negotiated includes both K_enc and K_mac where K_enc is the key used to encrypt traffic and K_mac is used to Hmac traffic.

For security, the client uses the public key of the server to encrypt the message.

When the server get the packets, he then uses his private key to decrypt the message. After getting the key, he will initialize Aes instance for this connection using K_enc, K_mac.

1.2 **Step2: Server reply agreed message**

They server then use the key to encrypt “agreed” message. The marshall procedure of the message will be discussed in section 2.

1.3 Step3: Client uses his device key to do authentication.

The device key of client is pre-stored in the server. Note for security purpose, the server stored te HASH of the device-key instead of plaintext. Also, the server encrypt the HASH form of the device key before he stores it in the disk to ensure security. How the server stores the message in disk will be discussed in section 2. How this key is established is omitted here.

The client encrypts his device-key and sends it to the server.

1.4 Step4: Server authenticate clients

the server decrypts it and HASH it. If the HASH result is the same with that stored in the disk, them the client pass authentication. Otherwise, the client will fail because of wrong device key.

1.5 Client send request to server

Client marshalls the request messages and sends it to the server.

1.6 Server response to request

Server unmarshalls the received messages, and responses accordingly.

2. Data Encryption

2.1 Client’s Encryption of Resource key

One very important point is: the resource encrypt key K_rsc should be the same for the connections authenticated using the same device-key, otherwise some connections will fail to decrypt the encrypted resource key.

K_rsc is generated using master-password with salt, and random initial vector. This key is generated only once and will not be changed later. Therefore, when the master password change, this key will not change.

The client encrypts the resource key before sending it to the server, and the server stored the encrypted version of resource key.

2.2 Encryption of traffic

K_enc is generated by a random string with salt. Note this key can be and should be different for each connection.

One important issue is whether the initial vector is kept the same for each single connection or not. According to [1], the initial vector should be different at each encryption to avoid revealing patterns. Therefore, for my implementation, the sender will send IV in plaintext along with marshall data to receiver. When receiver receives the packets, he will first get the IV to initialize the IV of his Aes instance, then unmarshall the data.

2.3 Server's Encryption of device-key

Server will first hash the device-key, and use AES/CTR to encrypt the hash form before stored it in the disk. The device-key is stored in a file called device-key. Note here the encrypt key will always be kept the same.

3. Data Marshall

3.1 Overview of data marshall and Unmarshall

3.1.1 Marshall

IV	Encrpted	msg	nonce	HMAC(msg+nonce)

```
encrypted = this.appendIV(this.aes.encrypt(this.aes.hmac.mac(this.appendNounce(data))), iv);
```

1) append the message with a nonce

2) append the HMAC of (msg+nonce)

3) encrypt the (msg + nonce + HMAC(msg+nonce))

4) append initial vector

3.1.2 Unmarshall

When the receiver receiver the messages, receiver will unmarshall it using the following procedure:

1) get IV, update the IV of Aes instance

2) decrypted messages, and get (msg + nonce + HMAC(msg+nonce))

3) unhmac the message to check integrity. He will HMAC msg + nonce to produce HMAC'(msg+nonce). If HMAC'(msg+nonce) is the same with HMAC(msg+nonce) which is appended in the message. Then the integrity is satisfied. Otherwise, the packets will be considered being modified.

4) remove the nonce and get the msg.

3.2 HMAC

The hmac key will be initialized with a random block. This key will be send together to the server during the key negotiation step. And this key will be kept unchanged during the connection.

3.3 NONCE

The sender will append a nonce to each message he sends, and the nonce he uses will be different each time. When the receiver receives a message that append with a used nonce, this packets will be dropped. To implement this, the server and client will each maintain a hash set to keep records of used nonce. The maximum value of nonce is set as Integer.MAX_VALUE. First, the nonce will be initialized to 0, and each time it is used, the value will increase 1, when the value increases to Integer.MAX_VALUE, the connection will be closed.

3. Resistant to attack

3.1 Reply attack

Since nonce is appended in the message, the replay packets will have the used nonce and will be dropped. Also, the attacker will not be able to modify the nonce, since the integrity of the packets is insured by HMAC.

3.2 Integrity

Since the packets are appended with his HMAC, the receiver can check the integrity of the packets. Therefore, integrity can be ensured.

4. Test Result

To performance reply attack, one API function is written to change nonce. This function should to removed later.

The testing result is as following:

```
Setting up the connection...

Launch server...

[Server]  set hashed deviceKey to '7c6a61c68ef8b9b6b061b28c348bc1ed7921cb53'

[Server] save deviceKey to Device Key



Client 1 try to connect server...
```

```
[Client] Negotiating with server...send encryptkey & hmacKey

[Server] processing negotiating packets, "HASHCODE" of pwd = COMM_PWD salt =
1486911496 ivBytes = 254313133

Negotiating success...

[Client] Authenticate himself to server by sending Device key: 'passw0rd'

[Server] authenticating clients, clients device key is: passw0rd

Authentication success...


Client 2 try to connect server...

[Client] Negotiating with server...send encryptkey & hmacKey

[Server] processing negotiating packets, "HASHCODE" of pwd = COMM_PWD salt =
2082781507 ivBytes = 1325342049

Negotiating success...

[Client] Authenticate himself to server by sending Device key: 'passw0rd'

[Server] authenticating clients, clients device key is: passw0rd

Authentication success...


Basic testing

Client 2 Changing the password

[Client] try to change  password to 'passw1rd

[Server] responsd: Success


Client 3 try to connect server...

[Client] Negotiating with server...send encryptkey & hmacKey

[Server] processing negotiating packets, "HASHCODE" of pwd = COMM_PWD salt =
1805366459 ivBytes = 997627950

Negotiating success...

[Client] Authenticate himself to server by sending Device key: 'passw1rd'
```

```
[Server] authenticating clients, clients device key is: passw1rd

Authentication success...


More tests...

Test replay attack...

Replay attack!


Checking wrong password

Client 4 try to connect server with outdata device-key...

[Client] Negotiating with server...send encryptkey & hmacKey

[Server] processing negotiating packets, "HASHCODE" of pwd = COMM_PWD salt =
1885534170 ivBytes = 1904100058

Negotiating success...

[Client] Authenticate himself to server by sending Device key: 'passw0rd'

[Server] authenticating clients, clients device key is: passw0rd

Wrong device-key throws CorruptMessageException


cleaning up

*** PASS ***
```

Reference

[1]Should I use an initialization vector (IV) along with my encryption?

http://stackoverflow.com/questions/65879/should-i-use-an-initialization-vector-iv-along-with-my-encryption