Comp 443 - Modern Cryptography

Mehmet Ustek

Project 2 Report

Completeness of the project:
All requirements are done regarding the homework description. Please follow the running instructions below.

Please be aware that code completely works, but it sleeps for 10 full seconds.
Running Instructions:
For the communication phase, run two parallel part1.py programs.
For man-in-the-middle, first run man_in_the_middle.py then run two parallel part2.py programs. Then call the init command **first in the man_in_the_middle.py. This flow has to be followed for program to work correctly.**
For running examples, please see **figures 1 to 5.**
Important Notice: If you want to re-run the programs, please delete all contents of txt files. I did not devise a strategy for that as it was not asked in homework. If these file contents are not deleted, then my program will not run. Always “init” man_in_the_middle.py first before doing “init” command to part2.py programs.
For both programs, first enter the user name for each user. Then type “init” to start the program. The program will wait for each initialization for 10 full seconds. After the program wants you to enter a message, enter the message.
The man_in_the_middle works as follows:

- The first user Alice sends her message.
- The attacker receives this message and decides what to send to Bob.
- When Bob receives the Attacker's message, it responds to it. The attacker again receives this message from Bob and decides what to send to Alice.

Thus, the man_in_the_middle lives in its own program.

Explanation of the Implementation and non-trivial Functions:
I used parameters g = 21 and prime order p = 16069 throughout this project. As the homework description suggests, both parties agree to use a predetermined prime and generator g. Although much larger values can be used, for faster convergence and comprehensiveness, these values are sufficient.

generate_secret_and_g_a (g,p): Given the generator g and prime order p, this function calculates the secret and the public key of the user. Namely, it randomly outputs a secret a, and calculates g^a modulo p.
calculate_private_key(y, secret, p): Given the secret and the public key of the second party, the first party calculates its private key. The following function for calculation is just like the one before, y^a modulo p = g^{ab} mod p.
encrypt(key, message, filename): Given the key, message and the filename, this function encrypts the message using AES-128 CTR encryption, with a random nonce. Then the function from Crypto.Cipher library encrypts the message and returns a nonce and a ciphertext. I wrote these two values to the file with **comma separation.** Therefore a message is of the form:

"Tz8Z0XGL+Gc=,82bqpy0YX9E=000000000000000" The first part until the comma is the nonce part and the second part after the comma is the ciphertext with 15 zeros appended at the end.
decrypt(message_input_from_file, key): Takes the message written to the file and splits it into nonce and ciphertext. Next, it decrypts the message using these two parameters with AES-128 CTR mode. It then returns the plaintext.
user1_key(K_ab_A): K_ab_A is the calculated private key, namely the g^{ab} mod p. This function returns the H(g^{ab} mod p) in an ascii form.

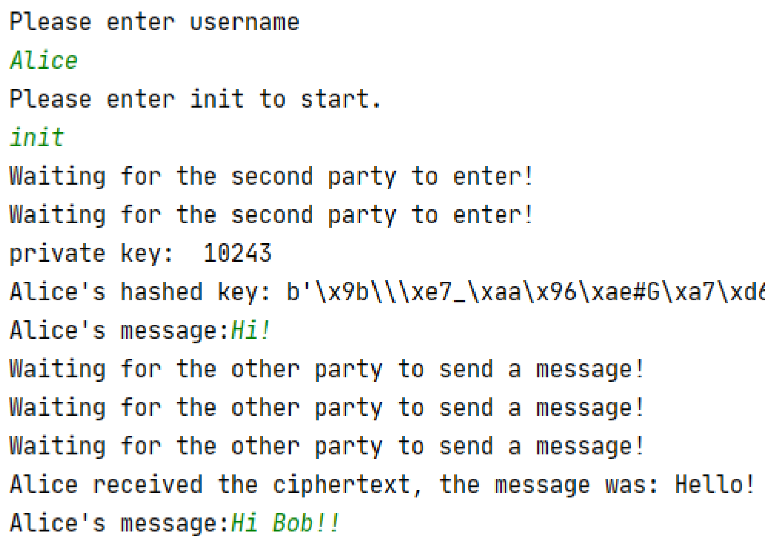
Having created all of these functions, first we have to initialize the communication file with public keys from both users. To do that I created two threads, since one user has to wait for the other user to write their username to start a session. The first threading function is get_user_input(file) and the second threading function is user2(file).

Get_user_input(.) function is the first person who wants to initialize the session. The function generates the secret and the public key for the user and writes this public key to the file. Next, it waits for another user to write their public key to the file. It checks the file every 10 seconds if there was another public key. In this case, since this user wants to communicate with another, this user's public key will be always written first to the file. Thus this user will always listen to the second line of the file. I used mutex locks for this function in case of any racing condition. Moreover, I used python built-in pow() function to calculate the y^a mod p, since this built-in function is a safe function for large values. In fact, initially I used classic representation (y**a) % p and did not understand why this was resulting incorrectly, until I figured out this representation was not safe.

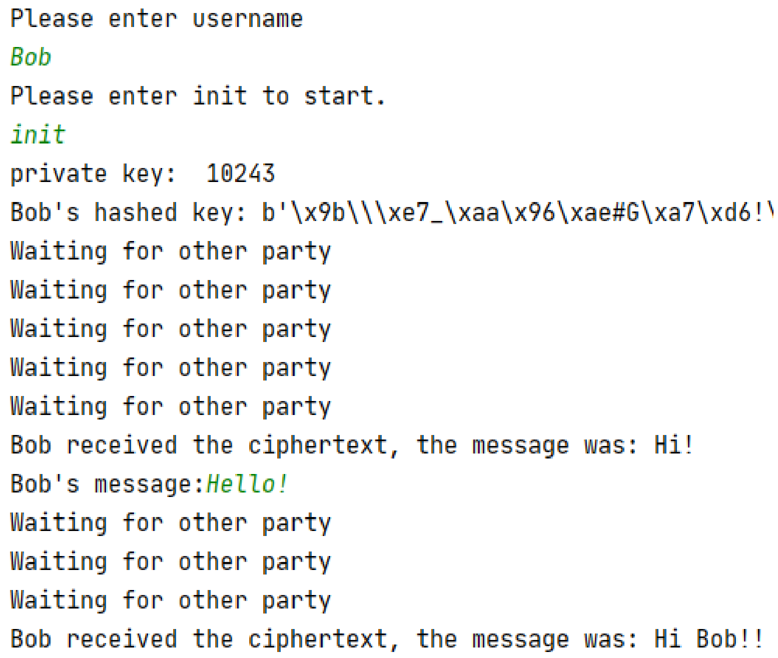
Next, the user2(.) function is pretty much the same as the get_user_input(.) function. I created two other functions to prevent race conditions for threads. In this case, the second user does not wait for any input from the first user and directly gets the public key of the first user to calculate g^{ab} mod p. Next, this user writes its public key to the file and the first user wakes up and continues with its thread. I also used mutex lock here to prevent race conditions.

After implementing these algorithm in a single-file manner, the TA Osman Biçer informed me that we do not need to have a single-program manner and we are expected to use one program for each user. Therefore, I updated the code to get rid of threads and other unnecessary functions to implement this strategy. The multiple program approach is more easy to do and the logic is similar. Instead of threads, we have two separate programs.

Now that we have private keys, we can start the communication phase. Here, the parameters are username1, username2, private key of the first user, private key of the second user and the file to read and write. At the beginning of this function, both users calculate their hashed private keys. This conversation goes on until one of the parties input "-1". The first user will start the conversation. The program will immediately encrypt this message and write the nonce and ciphertext to the file. The second party will read the message and this message will be displayed in the console. Next, the second party will send a message and this process will continue until someone inputs "-1". The file will be read every 10 seconds by the two parties. Alice will use her key to encrypt the file and Bob will use his key to decrypt and vice versa. The example of this conversation is as follows in Figure 1. Notice that after initialization, both parties receive their private key, and this key is displayed on the console for showing the work.



```
Please enter username
Alice
Please enter init to start.
init
Waiting for the second party to enter!
Waiting for the second party to enter!
private key:  10243
Alice's hashed key: b'\x9b\\\xe7_\xaa\x96\xae#G\xa7\xd
Alice's message:Hi!
Waiting for the other party to send a message!
Waiting for the other party to send a message!
Waiting for the other party to send a message!
Alice received the ciphertext, the message was: Hello!
Alice's message:Hi Bob!!
```



```
Please enter username
Bob
Please enter init to start.
init
private key:  10243
Bob's hashed key: b'\x9b\\\xe7_\xaa\x96\xae#G\xa7\xd6!
Waiting for other party
Waiting for other party
Waiting for other party
Waiting for other party
Waiting for other party
Bob received the ciphertext, the message was: Hi!
Bob's message:Hello!
Waiting for other party
Waiting for other party
Waiting for other party
Bob received the ciphertext, the message was: Hi Bob!!
```

Figure 1 and 2: Communication Phase

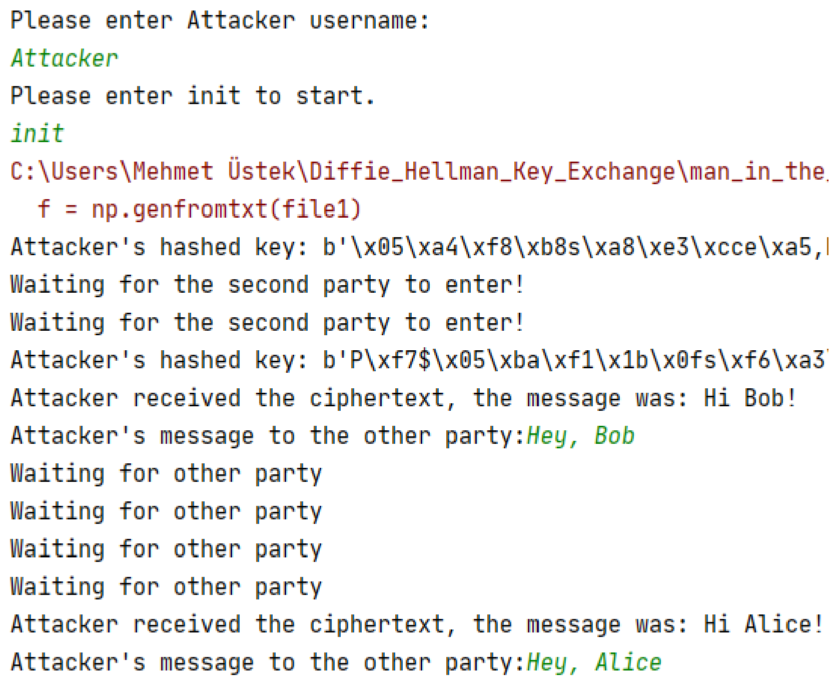
In the above figures, Alice is the first one to send a message, the message is decrypted with Bob's key and next, Bob sends a message. This iteration goes on until program termination.

Next, I implemented the man-in-the-middle function which takes file1 and file2 as parameters. The function first runs the 2-thread functions as before to trick the first user to generate a private key with the attacker. Then the attacker does the same with the second user and again generates a second private key with this user. Having both the private key this function calls the attacker_communication_phase(.) function which is similar to the communication_phase(.) function, with extra parameters. This function takes usernames for both parties, the attacker's private key for user 1, the attacker's private key for user 2, the user 1's private key with the attacker and finally, the user 2's private key with the attacker. Also two files that the attacker communicates with both users. In my case, I created “Communication_A.txt” and “Communication_B.txt” files.

After tricking the users to generate a private key with the attacker, this communication phase starts. The encryption and decryption process is just as before, however now there are 2 private keys, and 4 instances of these keys, 2 private keys for each user-attacker combination. The communication process is as follows:

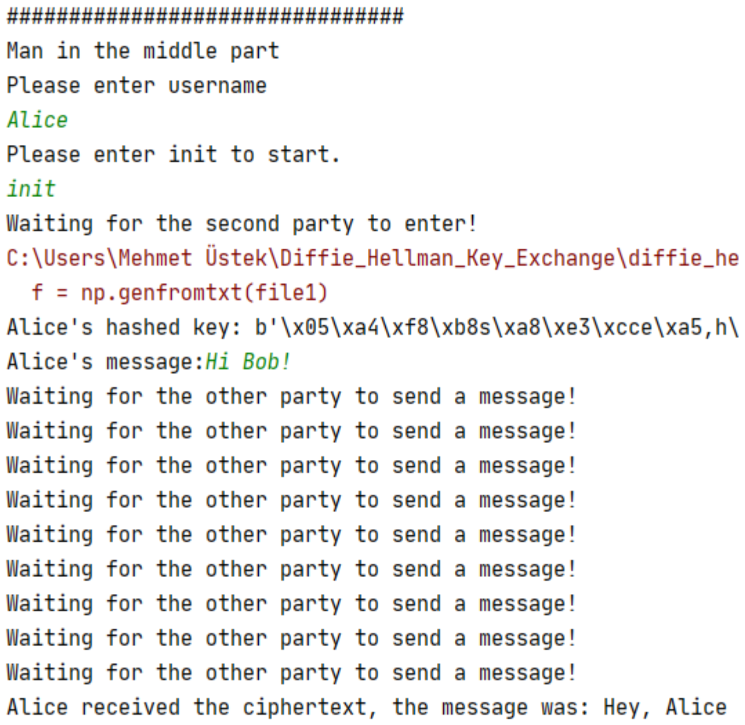
- The user 1 sends a message to the attacker.
- Attacker decrypts this message with its own key, and decides what message to send to the second party.
- The attacker sends a message to the second user with their own private key combination.
- The user 2 decrypts this message with its own private key, and send another message to the user 1.
- Again, the attacker decrypts this message and decides what to send to the first party.

Just as before, this conversation continues until one of the parties enter “-1”. An example of this communication is available on Figures 3, 4, and 5..



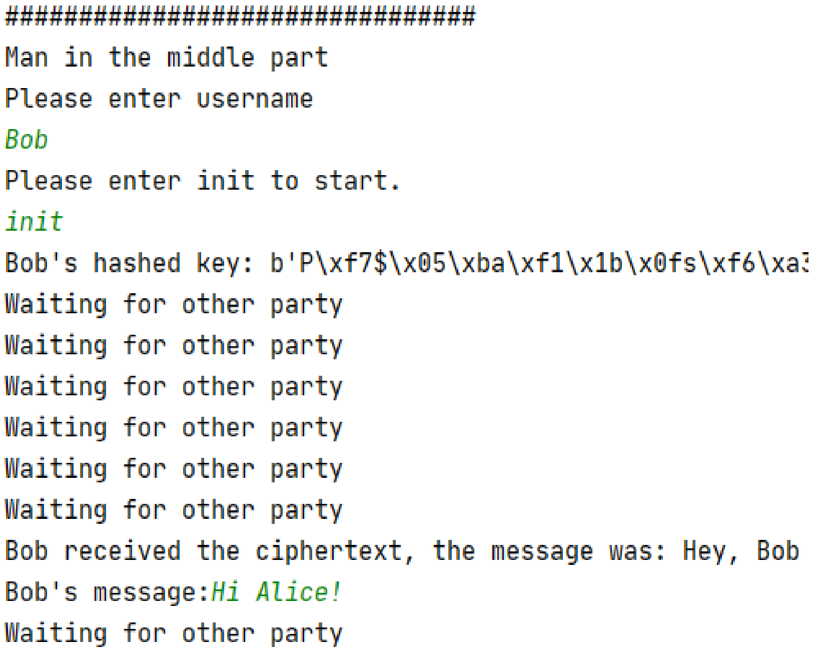
```
Please enter Attacker username:
Attacker
Please enter init to start.
init
C:\Users\Mehmet Üstek\Diffie_Hellman_Key_Exchange\man_in_the
  f = np.genfromtxt(file1)
Attacker's hashed key: b'\x05\xa4\xf8\xb8s\xa8\xe3\xcce\xa5,
Waiting for the second party to enter!
Waiting for the second party to enter!
Attacker's hashed key: b'P\xf7$\x05\xba\xf1\x1b\x0fs\xf6\xa3
Attacker received the ciphertext, the message was: Hi Bob!
Attacker's message to the other party:Hey, Bob
Waiting for other party
Waiting for other party
Waiting for other party
Waiting for other party
Attacker received the ciphertext, the message was: Hi Alice!
Attacker's message to the other party:Hey, Alice
```

Figure 3: Man-in-the-middle Attacker’s session



```
################################
Man in the middle part
Please enter username
Alice
Please enter init to start.
init
Waiting for the second party to enter!
C:\Users\Mehmet Üstek\Diffie_Hellman_Key_Exchange\diffie_he
  f = np.genfromtxt(file1)
Alice's hashed key: b'\x05\xa4\xf8\xb8s\xa8\xe3\xcce\xa5,h\
Alice's message:Hi Bob!
Waiting for the other party to send a message!
Waiting for the other party to send a message!
Waiting for the other party to send a message!
Waiting for the other party to send a message!
Waiting for the other party to send a message!
Waiting for the other party to send a message!
Waiting for the other party to send a message!
Waiting for the other party to send a message!
Waiting for the other party to send a message!
Alice received the ciphertext, the message was: Hey, Alice
```

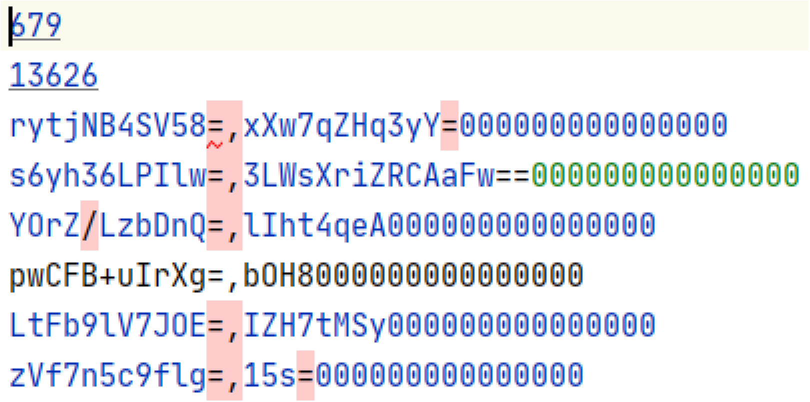
Figure 4: Man-in-the-middle Alice’s session



```
################################
Man in the middle part
Please enter username
Bob
Please enter init to start.
init
Bob's hashed key: b'P\xf7$\x05\xba\xf1\x1b\x0fs\xf6\xa3
Waiting for other party
Waiting for other party
Waiting for other party
Waiting for other party
Waiting for other party
Waiting for other party
Bob received the ciphertext, the message was: Hey, Bob
Bob's message:Hi Alice!
Waiting for other party
```

Figures 5: Man in the Middle - Bob’s session

An example of “Communication.txt” file is displayed in Figure 4. The first two rows are public keys for user 1 and user 2 respectively. The other rows are encrypted messages with nonce and ciphertext, separated with commas.



```
679
13626
rytjNB4SV58=,xXw7qZHq3yY=000000000000000
s6yh36LPIlw=,3LWsXriZRCAaFw==000000000000000
YOrZ/LzbDnQ=,lIht4qeA000000000000000
pwCFB+uIrXg=,bOH8000000000000000
LtFb9lV7JOE=,IZH7tMSy000000000000000
zVf7n5c9flg=,15s=000000000000000
```

Figure 6: Example of a conversation session on file “Communication.txt”

Acknowledgements:
I have never taken or given a code, or a snippet of code to/from anyone. Thus, all the work completely belongs to me.
I have also pushed all my code into github during the development of this project. In any case of suspicion of plagiarism, I can provide that repository.