Kathy Dai
John Alumbaugh
CSCE 4013: Cryptography Lab 1
Due Date: April 1, 2016

Structure of the Program

We have two scripts, representing a server and a client. The client assumed the Alice entity and the server assumed the Bob entity. We have also pre-generated RSA private and public keys for both Alice and Bob. Alice (client) has access to Bob's public key and not private key, and vice versa for Bob.

First, we have Alice generate an encryption key (line 73 in client.py) and encrypts it via RSA SHA-256 using Bob's public key and sent to Bob. Bob receives this key in a cipher text form and decrypts it with his RSA private key. We then calculate Bob's encryption key using the Diffie-Hellman protocol.

Within the Diffie-Hellman protocol, Bob generates a large random number, b, and calculates a value using the predetermined base and prime number, $bobValue = g^b \ mod \ p$. Then, we use pickle to pack up bobValue, the shared base, and the shared prime number and sends it to Alice. Alice receives this information (line 205 in client.py) and enters the Diffie-Hellman protocol. Alice mirror's Bob's actions by generating a large random number, a, and calculate $aliceValue$ using the given base and prime number. In Alice, we generate a temporary secret key where $tempSecretKey = bobvalue^a \ mod \ p$. We hash this temporary secret key with SHA-256 and create a long-typed secret key. We take the first 16-bits of this temporary secret key and save it as Alice's decryption key. The program then sends $aliceValue$ to Bob (line 151 in server.py). In the server script, we also create another temporary secret key using the same method as Alice: $tempSecretKey = aliceValue^b \ mod \ p$. We also hash this with SHA-256 and create a long-typed secret key. We take the first 16-bits of this temporary secret key and save it as Bob's encryption key.

We switch back to client.py where a signature key is randomly generated and then hashed, and RSA encrypted. We then send it to Bob. Bob receives this (line 220 in server.py) as Alice's encrypted signature key. We then decrypt this and save it as Bob's verification key. Whenever Alice's sends a message to Bob, she attaches a signature. Bob will use the verification key to determine that it is from Alice. We undergo the Diffie-Hellman protocol again and save the first 16-bits of the temporary key and save it as Bob's signature key and Alice's verification key.

For the last step, we send a 2000 byte message from Alice to Bob. We begin in the client script. The entire process consists of an RSA SHA-256 signing the plaintext with Alice's private key and hashing the signature, resulting in the hashed signature. We then encrypt the message using AES using Alice's encryption key, adding padding if necessary. We then use pickle to pack up the ciphertext and signature and send it to Bob. Bob receives the pair (line 230 in server.py) and attempts to decrypt and verify the integrity. We decrypt the message via AES and then remove any padding. We then verify the RSA signature using Alice's Public key. If the signature is verified, then it prints the plaintext and all the keys used to do this. Otherwise, it fails.

Then, from Bob to Alice, we send a 1000 byte message. We start in the server script. Instead of an RSA SHA-256 signature, Bob uses an HMAC SHA-56 protocol with his signature key to sign his message. We generate the ciphertext the same way using an AES encryption protocol. We then package the ciphertext and signature and send it Alice. Now in client, we receive the message (line 223 in client.py) and attempt to decrypt it in line 226. We decrypt the ciphertext via AES and then remove any padding. We then verify the HMAC signature using the Alice's verification key. If the signature is verified, it prints the plaintext and all the keys used. Otherwise, it fails.

What works and what doesn't?

I believe all works as intended. We used the PyCrypto libraries compatible with Python because most of the encryption and signature protocols are already programmed. Diffie-Hellman was done by hand. We also used sockets for server and client communication.