Assignment 3

Sergey Skovorodnikov, Poele, 35643097
Hannah Sarson, nobodysShoes, 47910138
Chris Dileo, zero, 27711035
Michelle Findlay-Olynyk

I. Github repo

https://github.com/cdileo/CPEN442_DeadIceCrow_VPN/tree/gui-chat-test2

II. Installation instructions

Our program is written in python 3 and uses the PyCrypto library. Unpack archived python files to the directory of choice, run using your python3 interpreter. Both computers will use same package of files but one must initially act as server and one as client.

III. Usage

Our program is entirely command-line driven. For starting client and server, provide the server and port details with their respective flags as indicated in the usage message. Notably, -S begins the machine in server mode, while -s allows you to speciy the ip of the server the client is looking for. Server will specify port for connection and client must specify both ip to connect to and port to connect on. Program will prompt for input of secret key and id during setup and output information at each stage of authentication. To confirm an entry, use the enter key. Once handshake is complete, chat is a simple matter of typing and hitting enter. The corresponding message will appear at your peer's console.

Once authenticated, client and server can chat using an encrypted channel. To protect PFS, a limited number of messages can be sent before the computers must re-connect (and establish a new key).

IV. Description of how our VPN works

To transmit data over the network, we use python's built-in socket functionality. Both the client and server will wait for data to arrive at their socket and then display it on screen after interpretation. The data is encrypted using PyCrypto's AES implementation with the CFB option. This produces a byte string that is then sent over the wire and decrypted on the other side.

We structured our mutual authentication according to the class slides on Diffie-Hellman; the server acts as Alice and the client as Bob. This initial handshaking uses encryption based off of the shared key provided. The server will send an initial message containing its user-generated id and a nonce challenge in the clear. The client will receive this message and send back a response consisting of the client's nonce and an encrypted message containing the client's id, server's nonce, and the client's half of the session key. The final step in the DH exchange is an encrypted response from server containing the server's id, the client's nonce, and the server's half of the session key.

We chose the DH exchange as it was a simple processs and were most familiar with it, therefore less likely to leave glaring flaws in our implementation.

Choosing the nonces is left up to python's built-in Random.getbytes. To choose the g and p values, we just took a good looking prime (i.e. pretty much arbitrary and small enough to work with). Talking to Kosta, he said not to worry about picking great primes as that's not really the point of the assignment. Choosing the secret values (a and b), was also left up to the Random.getbytes function and limited to 2 bytes as higher values would hang our systems.

We derive our session key (the one used to encrypt the messages once DH is complete) from the two half session keys from client and server. That is, using $g^a\%p$ and $g^b\%p$ to generate a single secret session key. The a and b values are generated anew on each start of the program and should ensure PFS. We did not have time to implement integrity checks and this will leave us open to cut-and-paste attacks.

V. Implementing this VPN in the real world:

We would likely use the same algorithms but improve our implementation overall. Adding in integrity checks to the messages would be essential, as would some sort of identification of who you're supposed to be talking to over this secure channel. Our modulus size would be much larger - we would choose proper (large, safe) primes for p and proper groups for g. We decided it was out of scope to worry that much about it in this assignment, however. As for encryption key size, we would use AES-256 just to be safe.

VI. Explanation of language and size of program

The program is written entirely in python 3 and uses mostly built-in libraries. The notable exception is the PyCrypto library which drives our AES implementation. The program is spread over 5 classes and in total is 628 lines of code, but could benefit from a refactoring as there is much duplicated between client and server. The interface is entirely command-line driven.