**Cyber Security Project 2**

Nimra Sharnez

**Results**

**Server:**

**A screenshot of a cell phone

Description automatically generated**

**Client:A screenshot of a social media post

Description automatically generated**

Above is the Server/ Client connections. The first connection is a successful connection with the right username and password combination and the second connection is an unsuccessful connection because it is the wrong username and password combination.

**Report**

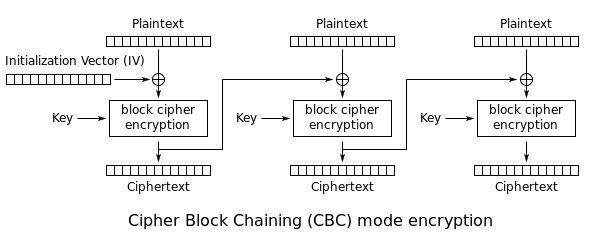
1. I hashed the password with the [hashlib](https://docs.python.org/3/library/hashlib.html) library using SHA-256. Before hashing, however, I created a salt with the [uuid](https://docs.python.org/3/library/uuid.html) library that helps generating a random unique ID. I used .hex() on the salt in order to convert any integers to a lowercase hexadecimal. Finally, I concatenated the salt with the entered password input (salt + password) and passed those two into the SHA-256 hashing algorithm. However, after some re-reading and online resources, it I found it is actually more common and normal to do password + salt, because we typically append the salt to the *end* of the password.

After creating the hashed password, I stored the username, salt, and hashed password in [passfile.txt](file:///Users/nimrasharnez/Desktop/nimra/server/passfile.txt) with space delamination. I chose the SHA-256 algorithm because it uses one-way compression. It is also very common in practice and one of the most trusted hashing algorithms.

For storage, we never stored the password in plain text, which makes the way I am storing secure. If a hacker accesses the [passfile.txt](file:///Users/nimrasharnez/Desktop/nimra/server/passfile.txt), they will be only able to see the username and salt in plaintext and the password would be hashed, so the plain-text would not benefit them in anyway as a username and it’s corresponding salt does not help in getting into someone’s account. Even if the hacker used a rainbow table to see common passwords hashed, the hashed password would not match to anything accurately on the table because it was hashed with a salt, not alone. This makes the storage secure.

1. I created public and private keys for the server with the help of the Crypto library in python. This is the [documentation](https://kite.com/python/examples/4916/crypto-generate-an-ssh-key-pair) I used to help generate the both keys. After generating the pair, I stored them in separate files, where the public key was accessible in the Client folder so they client could use it, and the private key was in the Server’s file as to keep it confidential from the client/ other potential “man in the middle-ers”.
2. I did the symmetric encryption with the passing of the session key by using the server’s public key to encrypt the randomly generated session key. This was done by importing the public key and using converting it into a [cipher object](https://www.dlitz.net/software/pycrypto/api/current/Crypto.Cipher.PKCS1_OAEP-module.html), from here, I then encrypted the cipher object public key with the session key using .encrypt(). By doing this, the server can decrypt the session key with their private key when they receive it. Now the client can send encrypted messaged to the server to decrypt with the session key they now have.

To encrypt the messages that the client sends (also the same way the server sends messages), I pad the message to send in order to make it an x amount of bytes, and then I use .encode(). I do the .encode() in order to make the string message into bytes (this is needed in order to make the following encrypting work). I then use AES, which requires the creation of an initialization vector. Because AES is fixed block size with 128 bites (16 bytes), the IV was created with a random generator that is 16 bytes in size. This IV was then XORed to encrypt the first block of the message, and that was encrypted with the session key to return a 16 byte cipher text. This cipher text was then used to XOR with the following plaintext block so on using AES from the [Crypto.Cipher](https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html) library.

The documentation I referred to can be found [here](https://www.dlitz.net/software/pycrypto/api/current/Crypto.Cipher.AES-module.html). The mode I used was **CBC**. I chose this mode because it made the most sense (visually) to me. It was simple and intuitive, and also given that there is a shared key between the client and server (the session key) that would only be obtainable by the server (since they can decrypt the session key with their private key), this seemed like a good option. A tradeoff here, however, is that because this required an IV, this exact IV would also need to be sent across to the server from the client. However, someone attacking who got a hold of the initialization vector in order to decrypt the message would also need to get a hold of the session key as well. Even if they did get the session key, they would need to decrypt it first before encrypting the message, which, again, is only possible with the server’s private key.

1. This would be secure because the messages sent back and forth are encrypted. They can only be decrypted with the decrypted session key. The way the server receives the session key is through its public, private key pair. This means that in order to decrypt the session key that is traded on a publicly visible network, only the server with their private key (never shared over network) can decrypt the session key in order to use to decrypt the messages.
2. No. Replay attacks don’t need to have any knowledge of what the messages actually are, they can simply just resend the encrypted messages to the server (as the client) in order to get into a user’s account. What I could do to prevent replay attacks is include timestamps in every message, where the server can expect any messages from the given session key in a five- or so-minute interval of the actual time. A problem with this is that it requires both the client and server to have synchronized clocks, but for the most part would suffice. This would work then because if the messages from the client were getting replayed, the Server would decline any messages from a replayed time outside of the five minutes of the actual time.
3. I learned how session keys can be transmitted on public channels so that only the intended recipient can decrypt it. I also learned more on how AES block ciphers work with both the XORing of an IV and also encryption with session key. I also found my ssh key on terminal which was cool and made me think about how Facebook handles the transmitting of sensitive information.

Something that clicked more for me was also salting and why they would work. Storing them in plaintext is smart because that would be the only way a server could compare the hashed password to the entered password. I also gained a deeper understanding of hashing and that it is not to decrypt, but to compare. Doing this project really helped in the confusion I had on why there are so many keys and other contributing factors in the handshake process. I also learned of all the handy libraries for even doing this client to server secure interaction. Lastly, when I click on my .pem file (which I also learned about from this project), I see the option to get a certificate for the key which further helped in understanding the subjects we were covering in class.

**Wire Share Screenshot**

**A screenshot of a video game

Description automatically generated**