Emulating SSL and Password Verification Project #3

Minhchau
Andre
Brennon
Paris
Dominic

This project was very straightforward by following the list of TODO functions. As a prerequisite to starting, we inserted users (same data as the write up examples) through the provided **add_user.py** function on the server side so there would be data to test. we also generated a public/private key pair to allow encryption / decryption of the session key during the initial handshake. It is also worth noting here that instead of using the PyCrypto library, due to inactivity, lack of support, and vulnerabilities listed here. We chose to use **PyCryptodome** instead for supporting libraries.

The first step was to generate a random AES key on the client side. The PyCryptodome library expects this session key to be either 16, 24, or 32 bytes long. Obviously, longer session keys provide stronger encryption but for the sake of this project, we went with 16 bytes. To generate the random AES key, PyCryptodome has a **get_random_bytes** function that generates it for us. But before sending the session key to the server side, it needs to be encrypted first with the server's public key for security. Again for the sake of this project and against better practices, the generated keys live inside of the **secret_keys/** folder at the root of the project. The server's public key was imported using **RSA** from Crypto.PublicKey and then the AES key was encrypted using **PKCS1_OAEP** and the public key. The encrypted AES key is then sent and received by the server where it uses the private key and **PKCS1_OAEP** to decrypt and obtain the plain text AES key. Upon decrypting the key, the server responds with an "Okay" signalling that it is now ready for encrypted usernames and passwords.

Before sending the message (consisting of username plus password as a string), it must be encrypted with the now shared AES session key. PyCryptodome conveniently has another function that does this for us called **AES** which takes in the session key as a parameter and then encrypts the message. But before being encrypted, the AES expects the plaintext to be a multiple of 16 bytes so we toss our message into the provided **pad_message** function and pass the result as our message to be encrypted. The message now can be securely sent over to the server where it uses the session key to decrypt the message. Taking the plain text message, our server begins authentication by parsing through the **passfile.txt** line by line looking for a match on the input username. If a match is found, we compare the stored hash password and input password by taking the input password + the matched user's salt value and returning a hashed value from the **hashlib** library. The returned hash value is then compared to the stored hash password value and if they are the same, the server returns the encrypted message "*User successfully authenticated!*" to the client. If a user is not matched or the hashes do not match, the server responds with the encrypted message "*Password or username incorrect*". Finally the client side prints the results from the server and closes it's connection with the opened socket and ends the session.

What we Learned

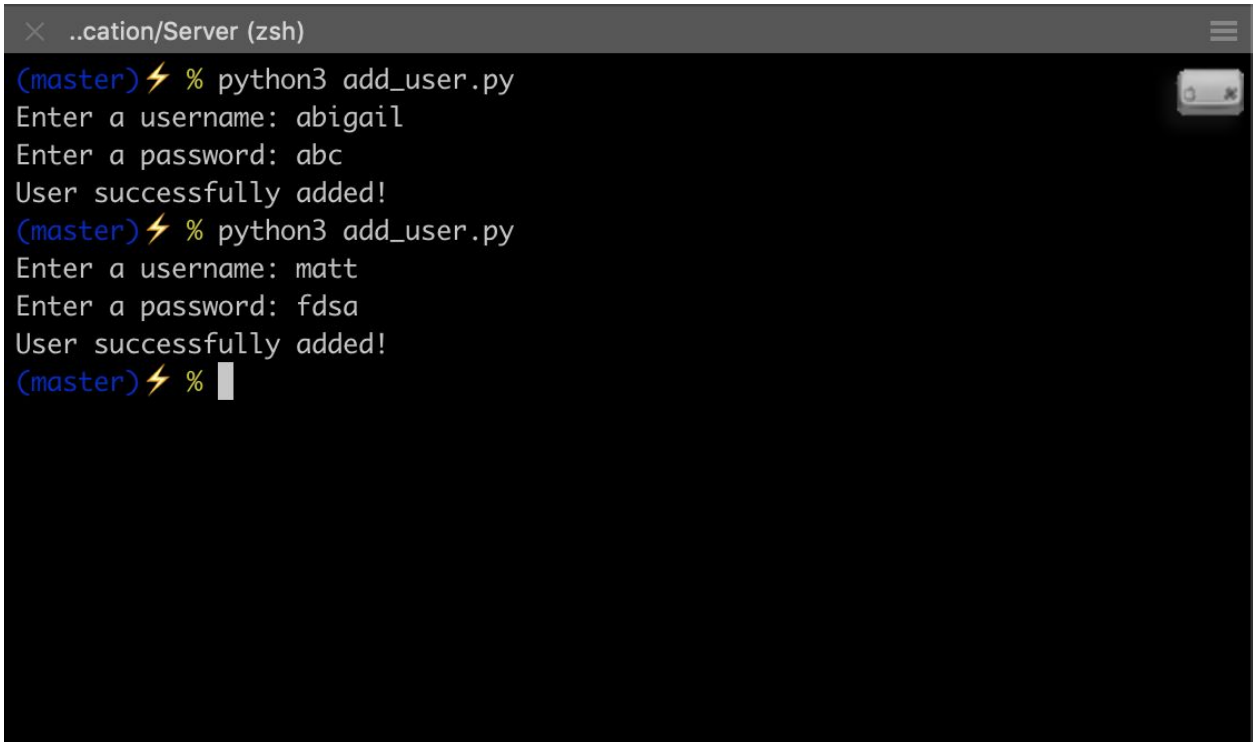
This project was a helpful demonstration of client-server interactions and public key cryptography. One of the areas of experimentation was in using new libraries that are important for crypto functionality. We used created AES keys with pyCryptoDome and checked the hash of keys using functions from hashlib. It is essential to understand the basic tools provided and their proper implementation in libraries such as these to ensure secure programs.

The other area of experimentation was in client-server communication. For some members of the group, networking experience is little to none, so learning about sockets and making connections on a localhost was very useful. It also served to show a small example of what professionals are doing out in the real world. By implementing SSL password verification, we got to see some of the inner-workings of modern network security.

Screenshots of Client-Server Interaction:

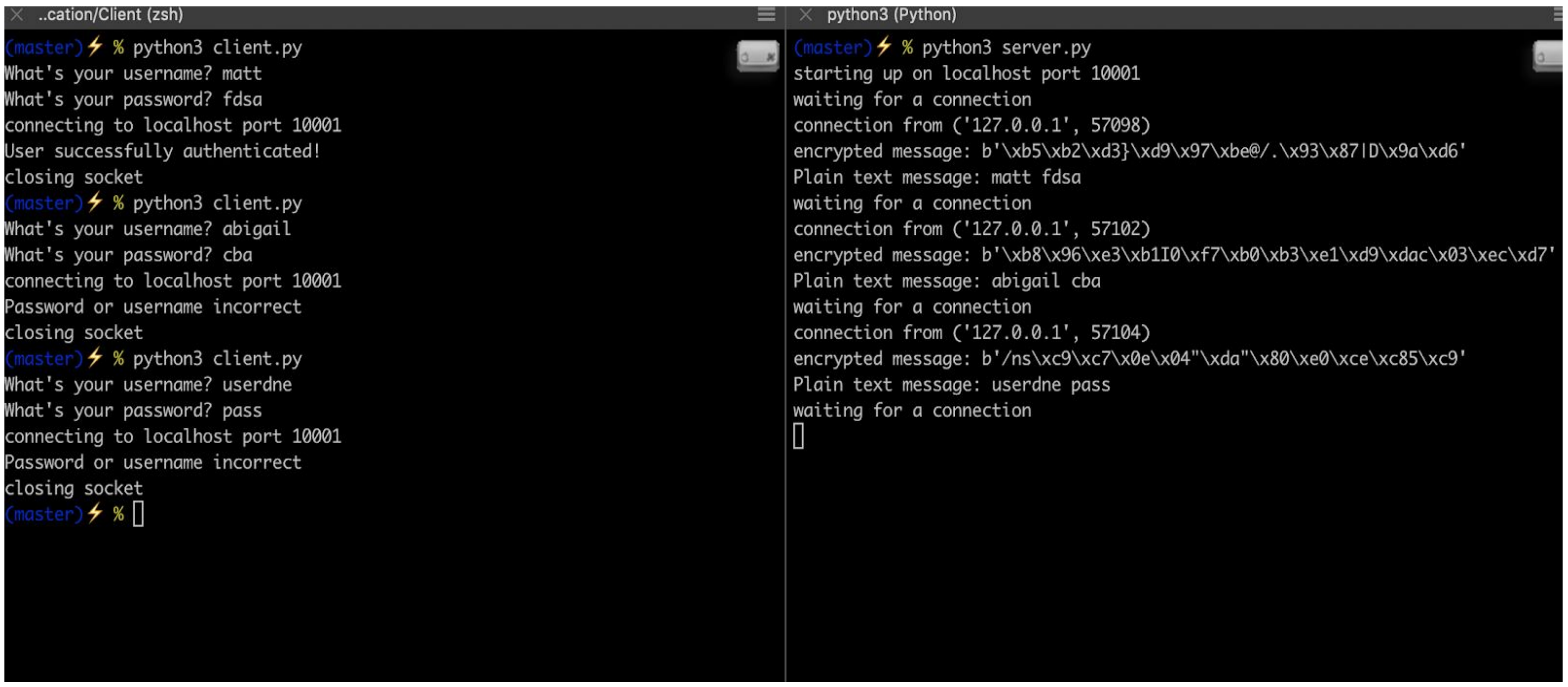
(used same interactions from the writeup.pdf)

Adding the users:



```
(master)⚡ % python3 add_user.py
Enter a username: abigail
Enter a password: abc
User successfully added!
(master)⚡ % python3 add_user.py
Enter a username: matt
Enter a password: fdsa
User successfully added!
(master)⚡ %
```

Client & Server Interaction:



```
(master)⚡ % python3 client.py
What's your username? matt
What's your password? fdsa
connecting to localhost port 10001
User successfully authenticated!
closing socket
(master)⚡ % python3 client.py
What's your username? abigail
What's your password? cba
connecting to localhost port 10001
Password or username incorrect
closing socket
(master)⚡ % python3 client.py
What's your username? userdne
What's your password? pass
connecting to localhost port 10001
Password or username incorrect
closing socket
(master)⚡ %
```

```
(master)⚡ % python3 server.py
starting up on localhost port 10001
waiting for a connection
connection from ('127.0.0.1', 57098)
encrypted message: b'\xb5\xb2\xd3}\xd9\x97\xbe@/.\x93\x87|D\x9a\xd6'
Plain text message: matt fdsa
waiting for a connection
connection from ('127.0.0.1', 57102)
encrypted message: b'\xb8\x96\xe3\xb1I0\xf7\xb0\xb3\xe1\xd9\xdac\x03\xec\xd7'
Plain text message: abigail cba
waiting for a connection
connection from ('127.0.0.1', 57104)
encrypted message: b'/ns\xc9\xc7\x0e\x04"\xda"\x80\xe0\xce\xc85\xc9'
Plain text message: userdne pass
waiting for a connection
```