Secure File Transfer Application

short line

**Group no. 6**

Ajay Dayma 17114006

Bhavye Jain 17114020

Kaustubh Trivedi 17114044

Shiva Reddy 17114045

Sai Krishna Abhiram 17114054

Ritik Kumar 17114063

Saurabh Singh 17114068

# **Contributions**

1. Ritik Kumar - (Pg. 7, 8, 11 - 13) - Contributed to the actual coding the implementation together with looking at the available libraries to handle encryption and key management. Contributed to the terminal based demonstration for the project.
2. Kaustubh Trivedi - (Pg. 7, 8, 11 - 13) - Contributed in the actual code, specifically connections.py and peer.py, where I handled the encryption and sending of data as bytes chunks from file. Contributed to make the tool interactive and demonstrate its POC.
3. Bhavye Jain - (Pg. 9, 10) - Created the overall design of the system encompassing the various entities, their roles and interactions. Evaluated the various options of signing certificates and decided on the hierarchy suitable for the project.
4. Saurabh Singh - (Pg. 4-5) Worked on securing file transfer using Sockets and developing an architecture that works similar to SSL/TLS security on the internet to ensure that communication is encrypted.
5. Ajay Dayma - (Pg. 2, 6) Worked on security issues of File transfer and how we can improve the existing solution. Explored different ways of securing communication, specifically JWT.
6. Sai Krishna Abhiram - (Pg. 14-16) - Packet Capture and Analysis using Wireshark
7. Shiva Reddy - (Pg. 2, 6) - Worked on what assumptions to take for our System. Explored different ways of securing communication, Specifically Symmetric Cryptography.

# 

# 

# **The Problem Statement**

*An organization needs an application which can help their employees to transfer files between them securely on the same network. Develop an application using socket programming to send files between two machines and secure the data transfer using a strong encryption algorithm. Capture these packets using a sniffing tool like Wireshark and show that data transfer is secure.*

# **Assumptions**

1. The required system is built to operate inside the network of an organization.
2. Since the clients belong to the same organization, they need not communicate to decide a cypher between them.
3. The certificates for secure communication are digitally generated by a master authentication server inside the organization. This server has the root certificate and key pair.
4. All the clients agree to encrypt data using asymmetric public-private key encryption.
5. The public ports of other nodes are already known within the network.
6. Every device entering our network already has the organization’s root certificate pre-installed on their devices, which would be used for key chain validation.
7. Auth server is online and it’s accessible to the user in the network.

# 

# Table of Contents

[**Contributions**](#_4lqp25cx7kth) **1**

[**The Problem Statement**](#_ovawxjoi07ia) **2**

[**Assumptions**](#_phlxkadp0igx) **2**

[**Securing the Connection**](#_9wyvcep6ipkk) **4**

[Security on the internet](#_8x0wu2xxg0oo) 4

[Security in our application](#_fa5nuburfk8c) 5

[**Other Methods explored to Secure the Connection**](#_ah3qjsdzyaq8) **6**

[**Solution Code Components**](#_sb86arti6oue) **7**

[Auth.py](#_mv6s1wbwrffz) 7

[Client.py](#_1nzy96a8fuhz) 7

[Peer.py](#_wp9uuagiuq8t) 7

[Connection.py](#_9pu6zrgl786b) 8

[Encrypt.py](#_cnanl41bb0gj) 8

[**Working of the System**](#_p7pd5378o8he) **9**

[**Working Demonstration**](#_wbzgpsyymzy1) **11**

[How to run our tool?](#_51v5unemw2fx) 11

[**Wireshark Captures**](#_mqitx5x0v5tp) **14**

[**Conclusion**](#_dyznryro1v1p) **17**

# 

# 

# 

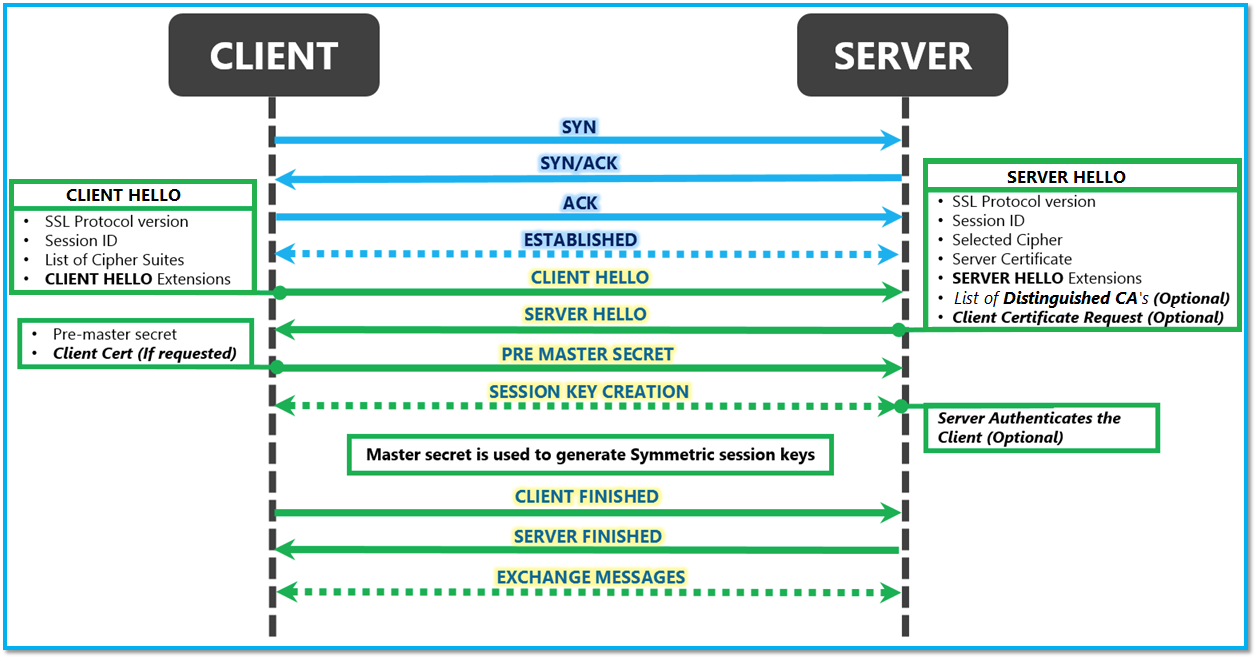
# **Securing the Connection**

## Security on the internet

Data is transferred over the public internet wherein it could easily be spoofed or sniffed if sent as plain text. TLS/SSL ensures that this transfer is encrypted using symmetric and asymmetric cryptography. Thus the data actually transferred is garbage to anyone else on the internet other than their intended recipient.

This is done by SSL Handshake and Certificate trust verification. This protocol dictates how both parties should interact to ensure trust and authentication. All this is carried out using an SSL Certificate. To generate that unique session key for encryption, both parties need to agree on particulars of the conversation. So this Handshake is that pre-conversation agreement. During the Handshake, the following things happen:

* Both parties agree on the protocol version
* Decide on the cypher suites to use
* Prove the identity of the server (and the client if required)
* Decide on the symmetric common session key for both parties



## Security in our application

Our application follows a very similar technique to SSL/TLS described in the previous section. Both the sender and receiver have a certificate issued by the authentication server. First, a TCP handshake is done between the two parties while creating a socket. After that, the following sequence of events happens to ensure that the session is secure.

* Client 1 Certificate Send: The client that wants to initiate a file transfer sends a request for a secure connection with the server. It sends it’s signed certificate to the other client.
* Client 2 Certificate Auth and Send: The receiver receives the transfer request, and verifies the authenticity of the certificate sent by client 1 using the auth server’s trust anchor certificate. It then sends back its own certificate to client 1.

The secure connection is thus established and their communication is encrypted now. Files and messages can now be sent that are encrypted by the public keys of the other client. It is decrypted by the private keys of the clients.

# 

# 

# **Other Methods explored to Secure the Connection**

Symmetric Cryptography:

Symmetric key cryptography (or symmetric encryption) is a type of encryption scheme in which the same key is used both to encrypt and decrypt messages. If the encryption scheme is strong enough, the only way for a person to read or access the information contained in the ciphertext is by using the corresponding key to decrypt it. The security of symmetric encryption systems is based on how difficult it is to randomly guess the corresponding key.

Asymmetric encryption takes longer to execute because of the complex logic involved, but allows to transfer data to users newly added in the network. Symmetric encryption, being fast, comes at the cost of the assumption mentioned below.

In our System, we are using asymmetric encryption because for symmetric encryption it is necessary for all nodes to have the key associated with all. We have not taken this assumption for a more robust system.

JWT(JSON WEB TOKEN)

This is one another way of secure communication using json objects. JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information.

This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.  
  
In basic terms, JWTs allow you to send data signed by an authority. Thus, this provides proof for the authenticity of the data. We can use JWTs in Asymmetric Encryption to share the Public Keys of the clients.

# 

# **Solution Code Components**

Complete code of the tool: [**https://github.com/codekaust/Secure-file-transfer**](https://github.com/codekaust/Secure-file-transfer)

## Auth.py

This is the master authentication server for the organization. It verifies clients and digitally signs their public key to generate certificates for the clients. This server has its own root certificate and private key. This functions as the trust anchor for all the certificates in the organization.

The server accepts HTTP requests from a client. It verifies the encrypted data provided by the client (username, password) and if these fields are valid, it proceeds to generate the certificate. The server checks if the client is registered in the organization and if successful, it signs the client’s public key and responds to the client with a certificate. The server uses **SHA256** hashing to encrypt the certificate. The certificate is generated in the **X.509** format.

## Client.py

This is the code which represents individual clients in the organization’s network. A user can use this code to safely log into the organization and get his key certified. He can then request or send encrypted files as required.

## Peer.py

This file stores the state of a known connected peer. This is particularly useful to store received public keys so as to encrypt all the traffic that is to be sent to that peer. This file is also responsible to send and receive data from the connected peer. These are done using TCP packets. We follow our own application layer packet protocol where the data is sent in the form of:

*!4sL<payload\_length>s*

Here, the 1st 4 bytes represent message type. We have the following message types:

REQUEST\_FILE = 'RQFL'

RESPONSE\_FILE = 'RSFL'

SEND\_CERT = 'SECE'

CERT\_RESPONSE\_VALID = 'RSCV'

CERT\_RESPONSE\_INVALID = 'RSCI'

Next 4 bytes represent payload length

The final <payload\_length> bytes are the actual data transferred.

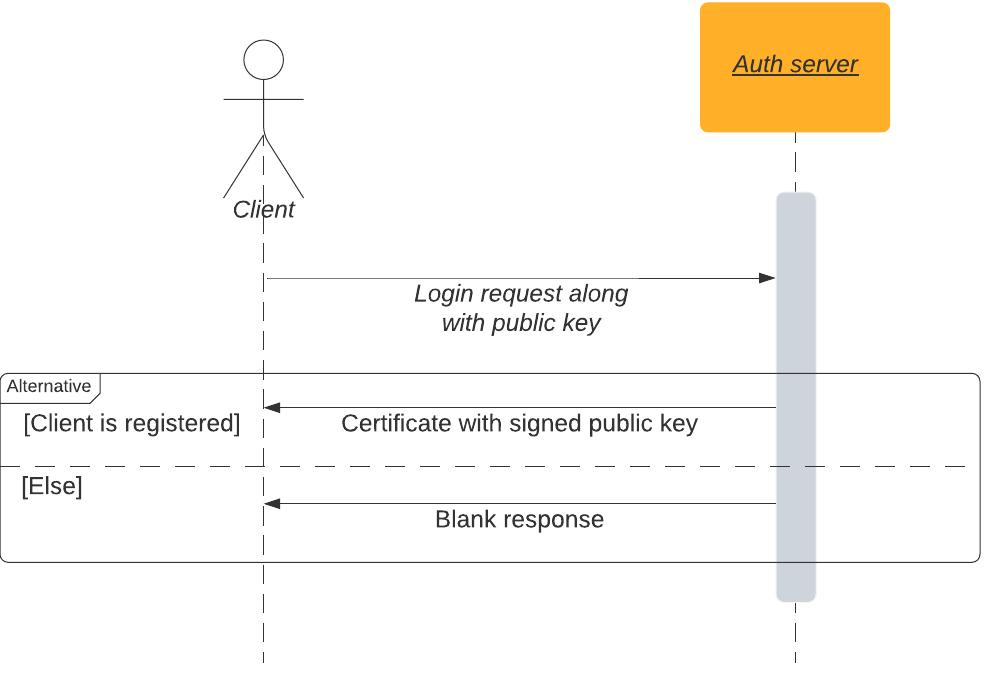
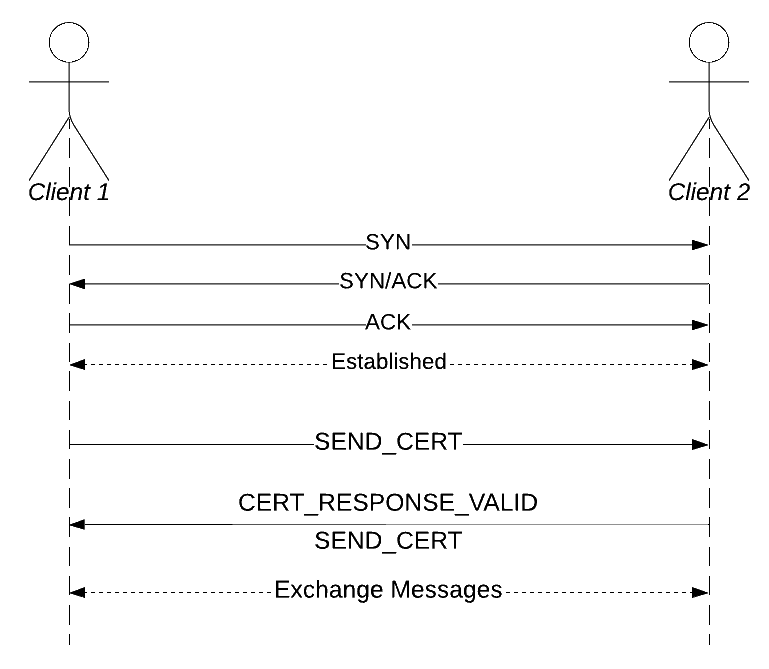
## Connection.py

This file house procedure for requesting files and handlers for handling incoming connections and messages based on the message type.

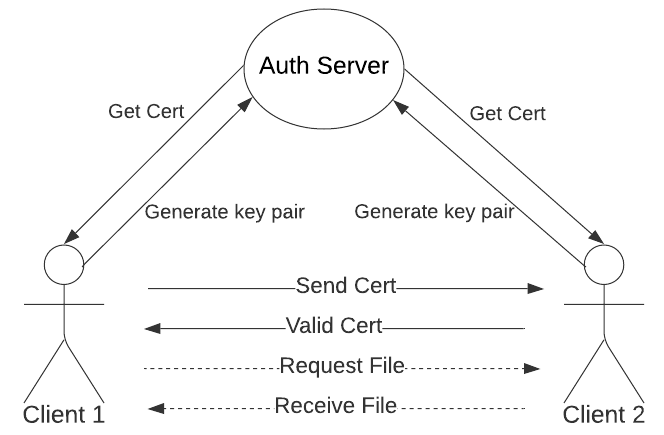
## Encrypt.py

This file contains a helper class and functions to handle and generate keys and certificates.

# **Working of the System**

To request or transfer files, a client must first log into the organization. The client sends a login request to the authentication server with its username, password and public key encrypted using the Auth server’s public key. The authentication server checks if the client is registered in the organization and if it is registered, the server generates a certificate by signing the public key of the client and adding other information. In case of any discrepancy, a 401 unauthorized message is sent.

When a client who is logged in wants to request a file from another client, a TCP connection is first established between the two clients via a 3-way handshake. SYN and ACK messages are exchanged to establish a TCP connection.

Once the connection is established, client1 requesting the file sends its certificate to the client2. The certificate contains the public key of client1. Client2 checks the validity of the certificate and if the certificate is valid, it responds with a certificate valid response and also sends its own certificate. Client1 verifies the certificate sent by client2 and if it is valid, a secure connection for file transfer is established. Now, client2 encrypts the file using the public key of client1 and sends it through the TCP connection.

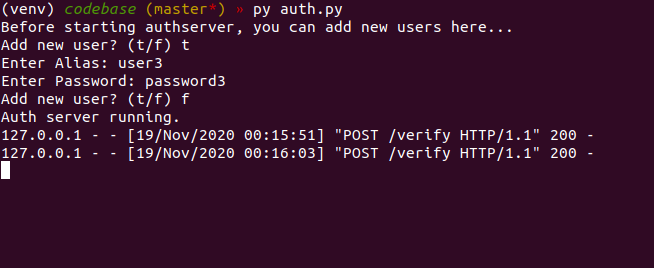
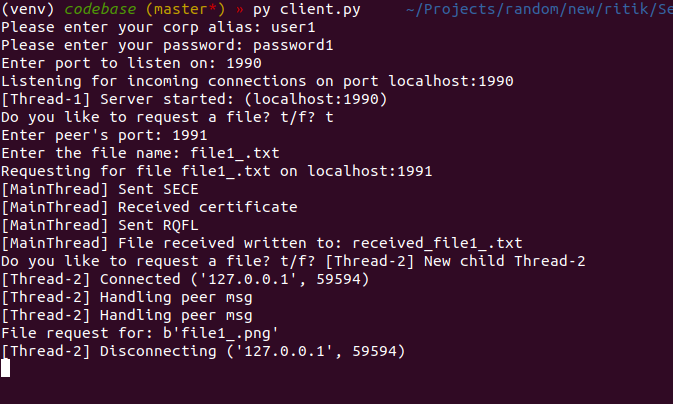
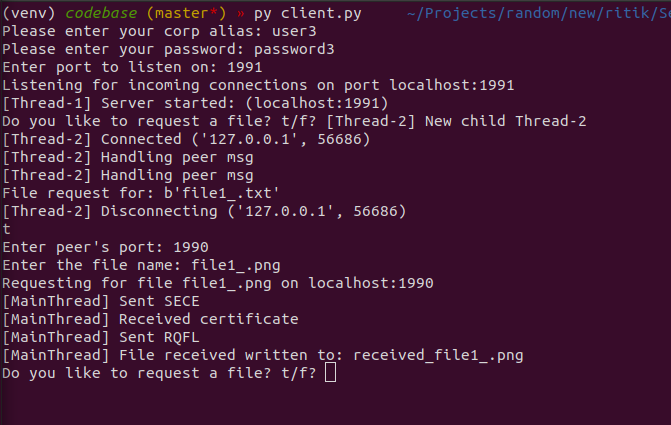
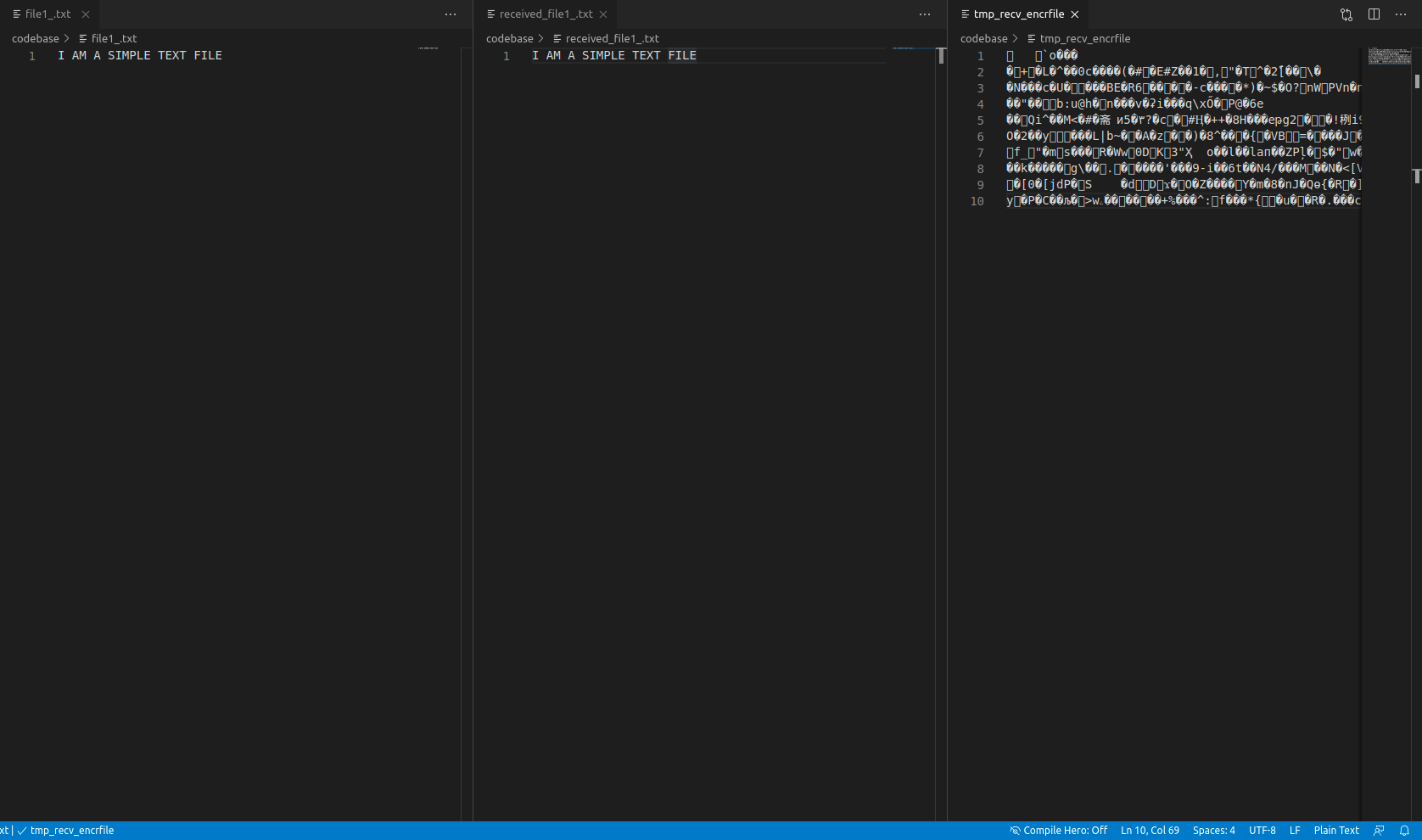
The above image shows the system as a whole and the interactions between the various entities. The auth server behaves as the master service in the system and the clients rely on the auth server to sign their public keys and declare the client trustable.

# 

# **Working Demonstration**

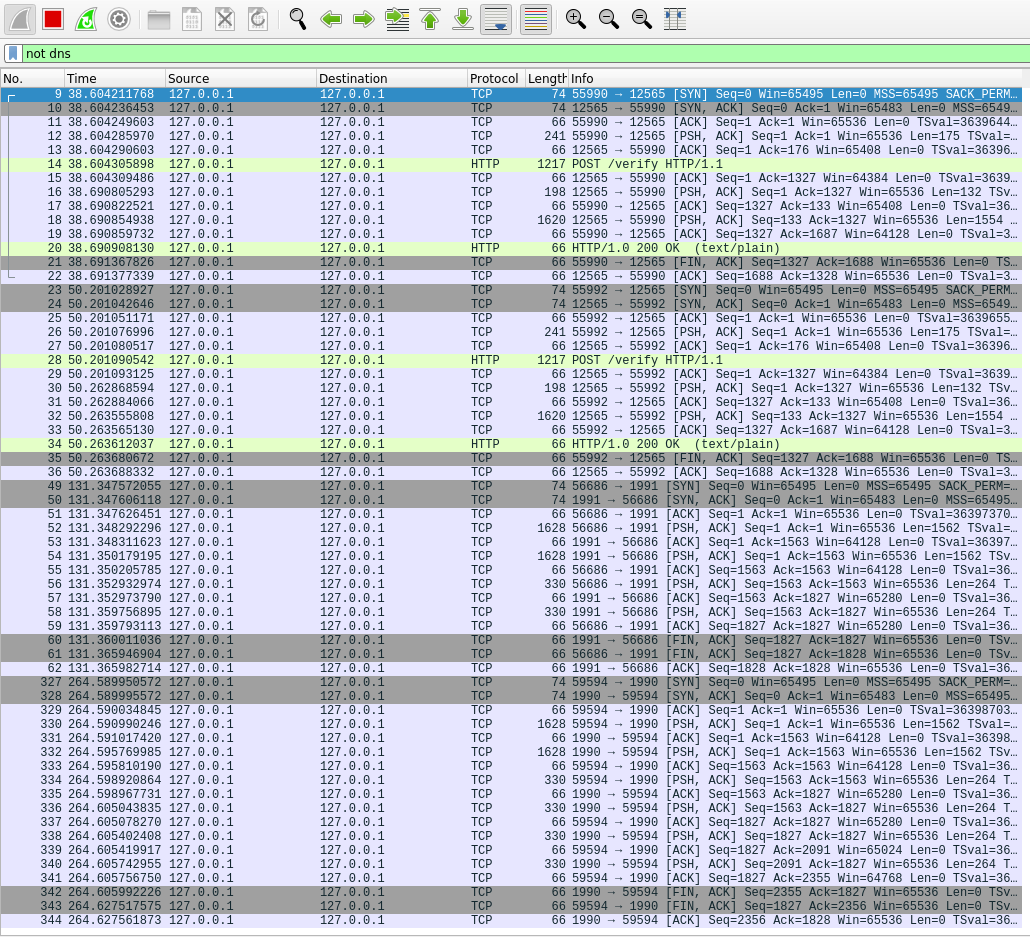
## How to run our tool?

NOTE: For easy usage and testing, our tool is configured to be run on localhost. Thus, the interactive terminal will ask you only for ports and not IPs. Port 12565 (randomly chosen) is reserved for the auth server.  
  
Following commands can be used to run the tool on a Linux machine (the tool is not platform restricted though):

1. Clone repository and change directory
   1. cd Secure-file-transfer
2. Install Virtual Environment
   1. sudo apt install virtualenv
   2. virtualenv -p python3 venv
   3. source venv/bin/activate
3. Install Dependencies
   1. pip install -r requirements.txt
4. Run Auth Server
   1. python auth.py
      1. This will start a python authentication server on port 12565. This server will be responsible to provide certificates to clients when they provide correct user alias and password.
   2. Firstly, this prompts if you want to create some new clients. Already created clients are: {“user1”: “password1”, “user2”, “password2”}. You can create clients using this prompt (press ‘t’ to create client).
   3. Once all the clients are created you can press ‘f’ to start the auth server.
5. Run Clients
   1. python client.py
      1. This will start a python client (a peer which can receive and send files).
      2. This python client asks for user alias and password and contacts auth server to get these verified.  
         Once verified, the client will receive a certificate corresponding to its public key.
   2. You have to start two clients, let's say using user1, user3 on ports 1990 and 1991 respectively. Now, to request a file, press “t” and enter.   
      Enter port number as the port number of the other client and the file. The file name can be relative to the client.py’s location or be the full name of the file.
   3. The file received is kept in the same directory as client.py with name as “received\_<original\_file\_name>”
6. This process is represented in the following screenshots:  
     
   *auth.py*  
     
     
   Here, we have started the auth server and created a user with credentials: user3, password3.  
     
   *client.py*In the first client, we have first served a file ‘file1.png’ request from the second client. Then requested file ‘received\_file1.png’. This file was then saved in received\_received\_file1.png  
     
     
   In the second client, we first requested a file ‘file1.png’ and the received file was saved in ‘received\_file1.png’. This file is then requested by the first client later.  
     
   Here, file1\_.txt was sent and received\_file1\_.txt was received. The encrypted data received is shown in file tmp\_recv\_encrfile.  
     
   

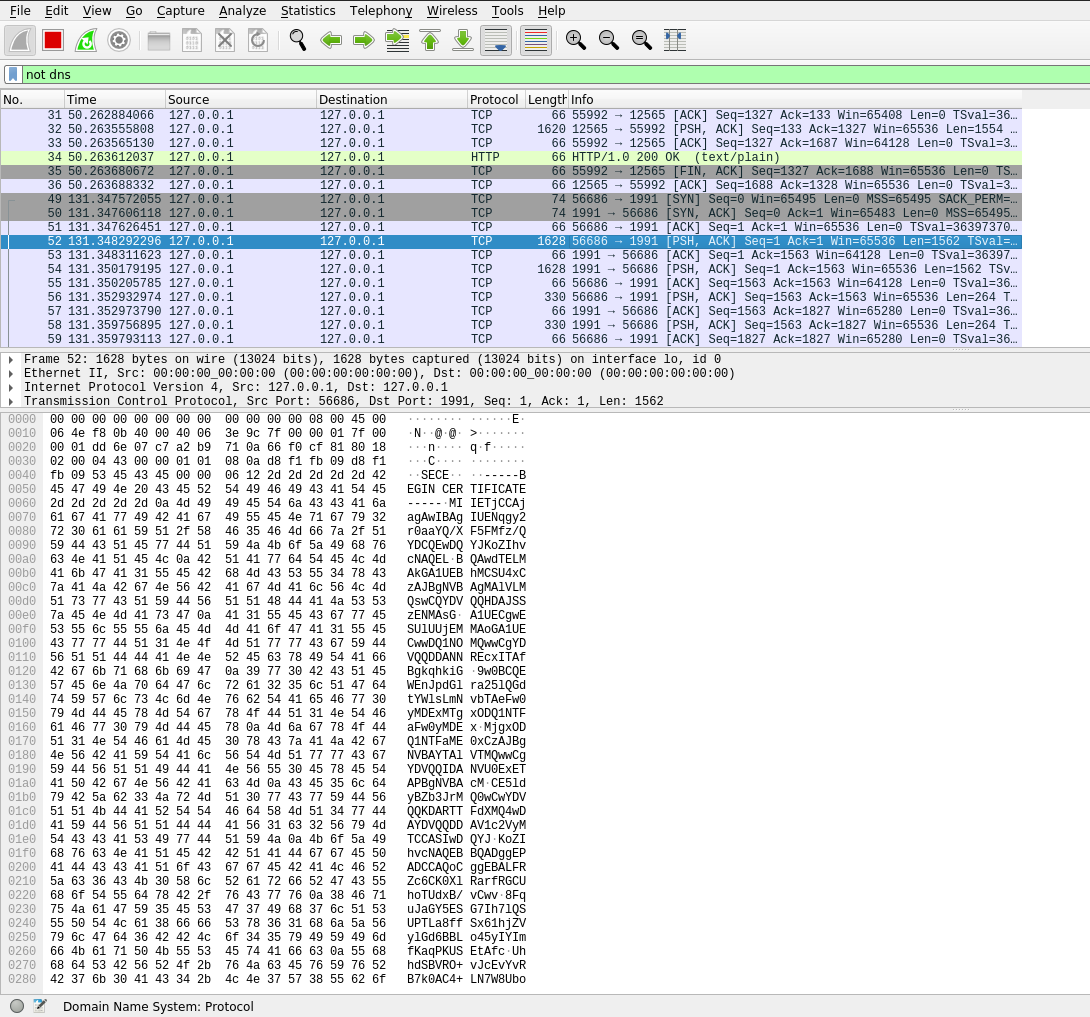
# **Wireshark Captures**

In this section, we capture TCP packets for the connection between two clients and prove that the data is being transmitted securely via encryption.



This image shows the Wireshark capture during a message transfer between 2 clients in our system. At first, a TCP connection is seen being established via a 3-way handshake using SYN, SYN/ACK and ACK.

Packet 36 payload contains the certificate of the client being monitored, as plain text. The certificate contains the public key for the client. The image below shows the details of the packet as viewed in Wireshark. Note the “BEGIN CERTIFICATE” in the data payload.



The following image shows a data packet for the message transfer. The payload length is 264 byt v es and the payload details section clearly shows that the data has been encrypted and nobody other than the 2 clients can know what is contained in the payload of the packets. Hence, we can say that we successfully established a secure connection for file transfer between the two clients.

# 

# 

# 

# Conclusion

From this project, we see that TLS-SSL provides for secure communication between two entities on the internet and its features can be leveraged to build a secure file transfer system for an organization.

We succeeded in creating the required system and tested it against various files. The system showed promising results with possible improvements to scale with the size of an organization. We were also able to show that the transmission is secure through a sniffing tool called Wireshark.

short dash