Secure File Transfer Application

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

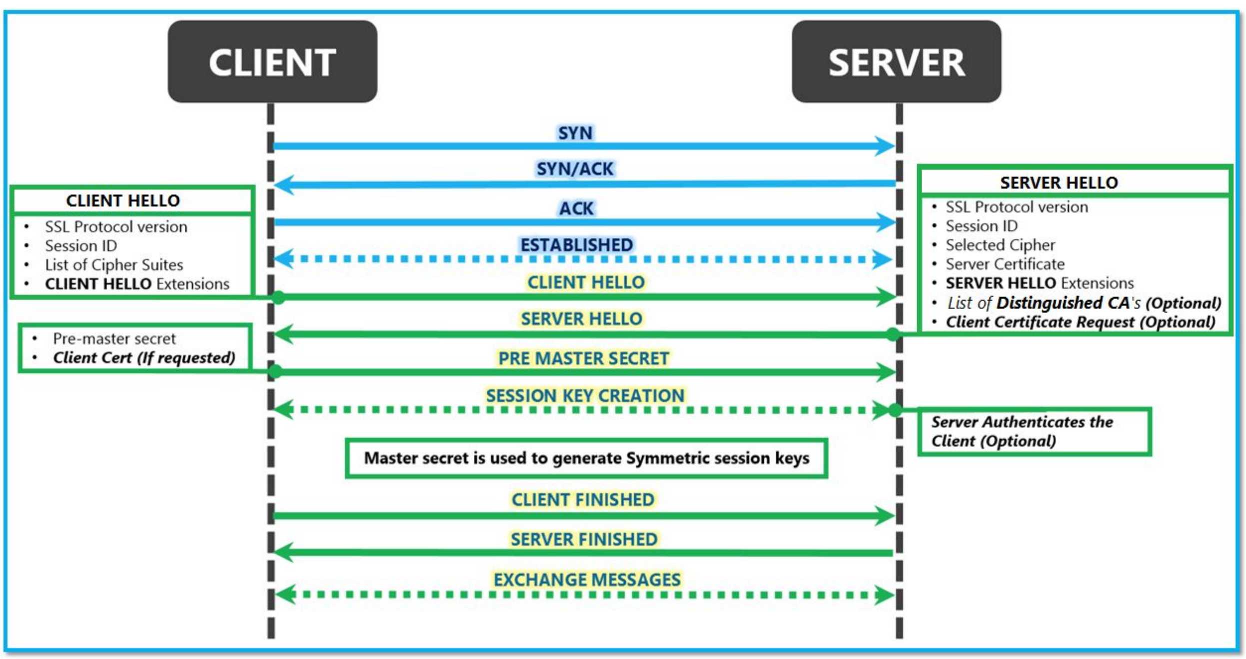
Securing the Connection

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

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.

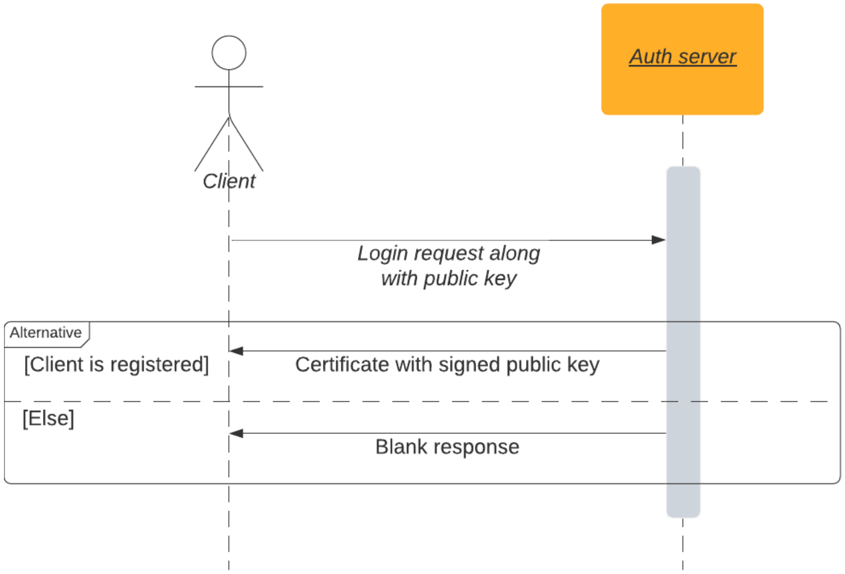
Connection.py

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

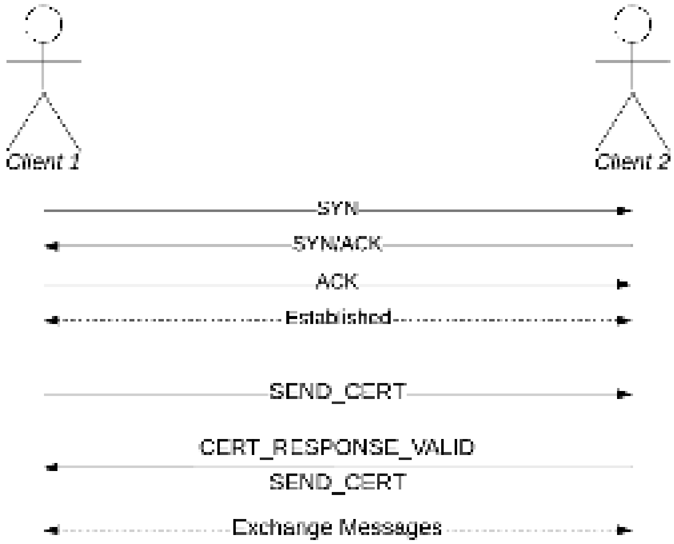
Encrypt.py

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

Working of the System



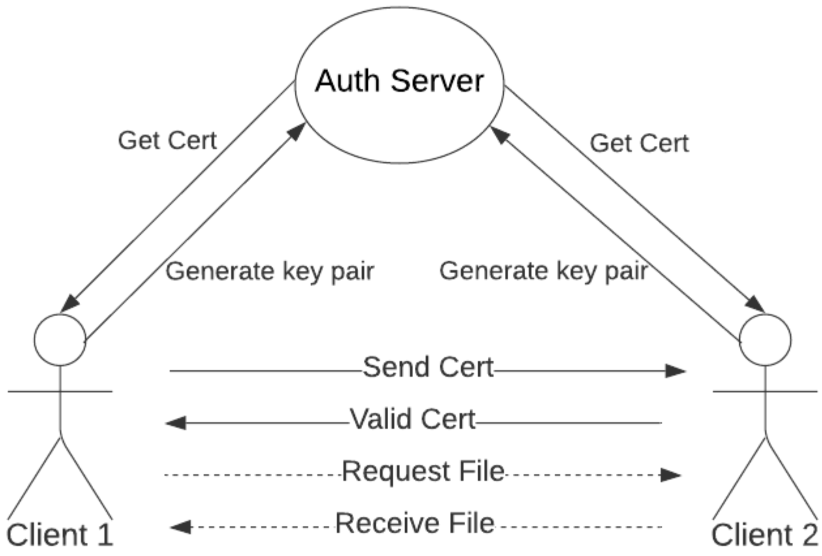
To request or transfer files, a client must first log into the organization. The client sends a login request to the authentication server with its username, password and public key encrypted using the Auth server's public key. The authentication server checks if the client is registered in the organization 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
    a. `cd Secure-file-transfer`
2. Install Virtual Environment
    a. `sudo apt install virtualenv`
    b. `virtualenv -p python3 venv`
    c. `source venv/bin/activate`
3. Install Dependencies
    a. `pip install -r requirements.txt`
4. Run Auth Server
    a. `python auth.py`
        i. 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.
    b. 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).
    c. Once all the clients are created you can press ‘f’ to start the auth server.
5. Run Clients
    a. `python client.py`
        i. This will start a python client (a peer which can receive and send files).
        ii. 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.
    b. 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.

c. 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

```
(venv) codebase (master*) » py auth.py
Before starting authserver, you can add new users here...
Add new user? (t/f) t
Enter Alias: user3
Enter Password: password3
Add new user? (t/f) f
Auth server running.
127.0.0.1 - - [19/Nov/2020 00:15:51] "POST /verify HTTP/1.1" 200 -
127.0.0.1 - - [19/Nov/2020 00:16:03] "POST /verify HTTP/1.1" 200 -
```

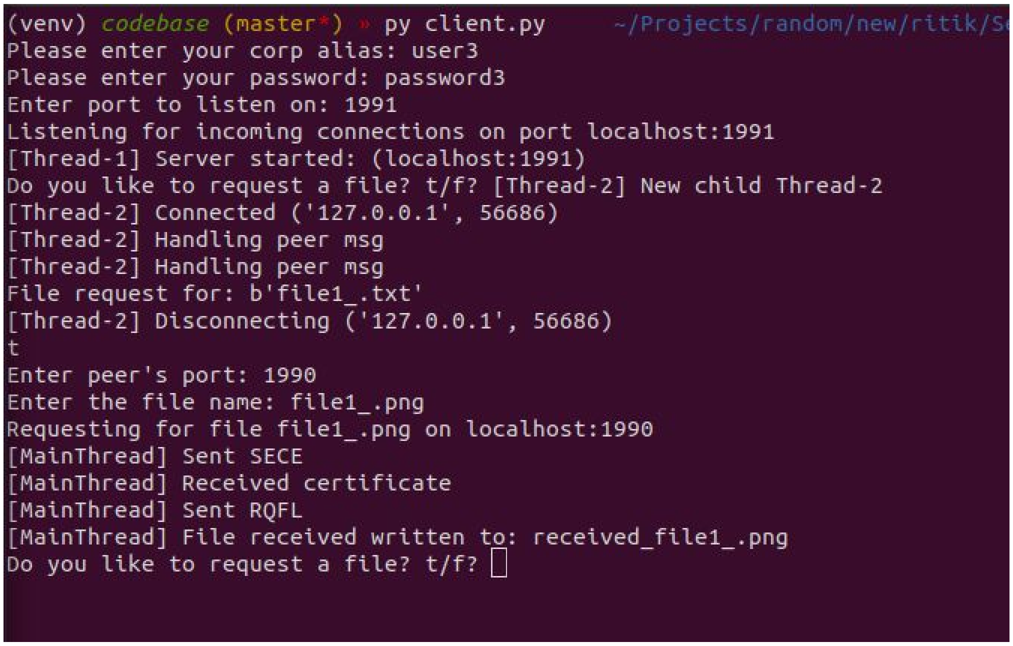
Here, we have started the auth server and created a user with credentials: user3, password3.

client.py

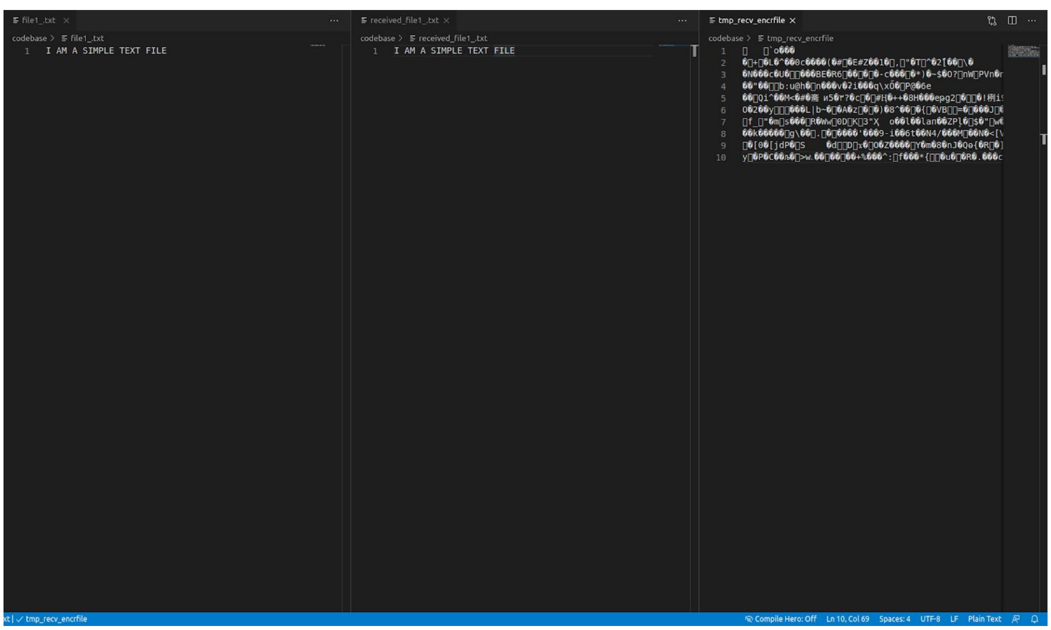
In the first client, we have first served a file 'file1.png' request from the second client. Then requested file 'received_file1.png'. This file was then saved in received_received_file1.png

```
(venv) codebase (master*) » py client.py      ~/Projects/random/new/ritik/Se
Please enter your corp alias: user1
Please enter your password: password1
Enter port to listen on: 1990
Listening for incoming connections on port localhost:1990
[Thread-1] Server started: (localhost:1990)
Do you like to request a file? t/f? t
Enter peer's port: 1991
Enter the file name: file1_.txt
Requesting for file file1_.txt on localhost:1991
[MainThread] Sent SECE
[MainThread] Received certificate
[MainThread] Sent RQFL
[MainThread] File received written to: received_file1_.txt
Do you like to request a file? t/f? [Thread-2] New child Thread-2
[Thread-2] Connected ('127.0.0.1', 59594)
[Thread-2] Handling peer msg
[Thread-2] Handling peer msg
File request for: b'file1_.png'
[Thread-2] Disconnecting ('127.0.0.1', 59594)
```

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.

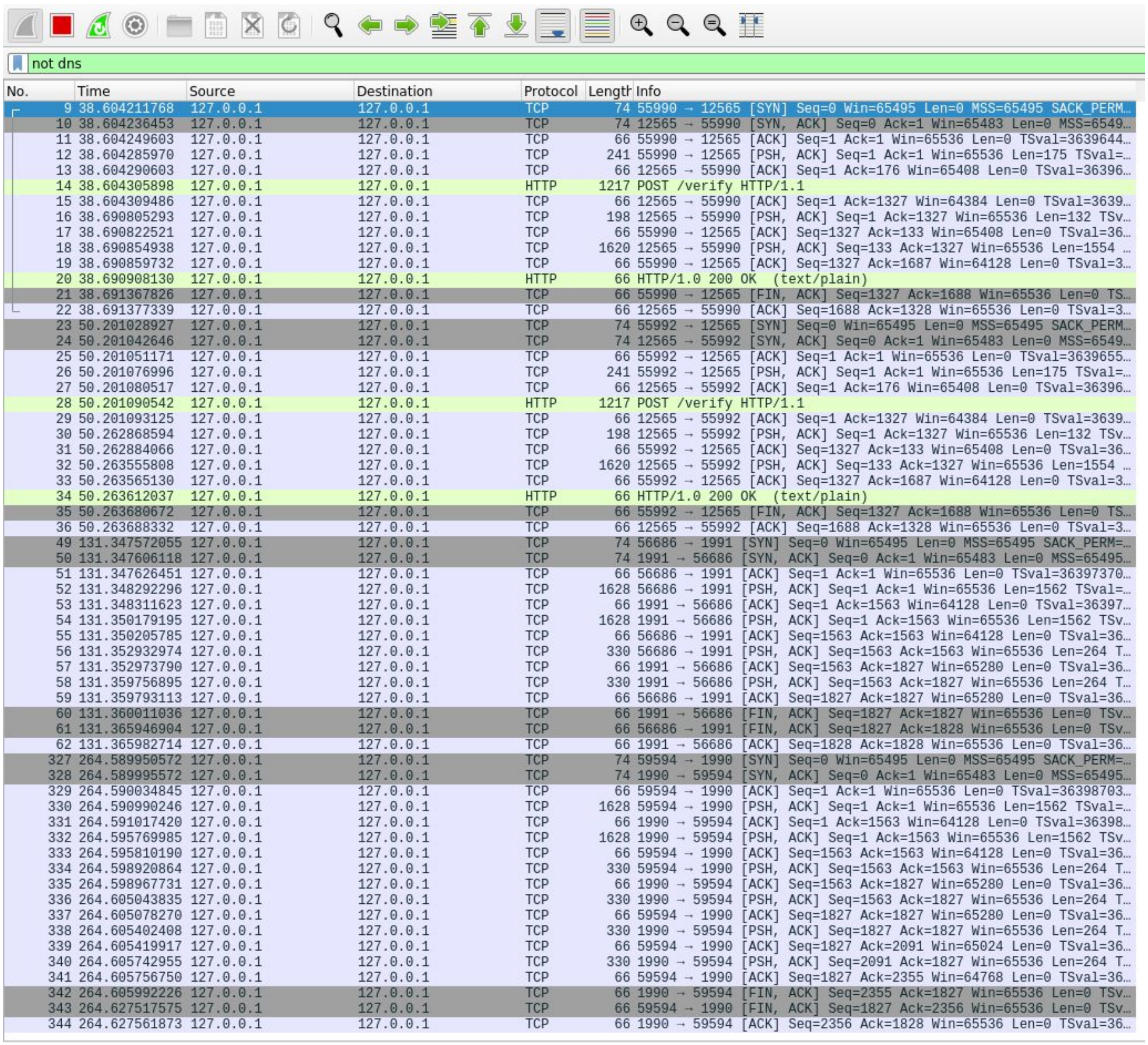


Here, file1_.txt was sent and received_file1_.txt was received. The encrypted data received is shown in file tmp_recv_encrfile.



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.

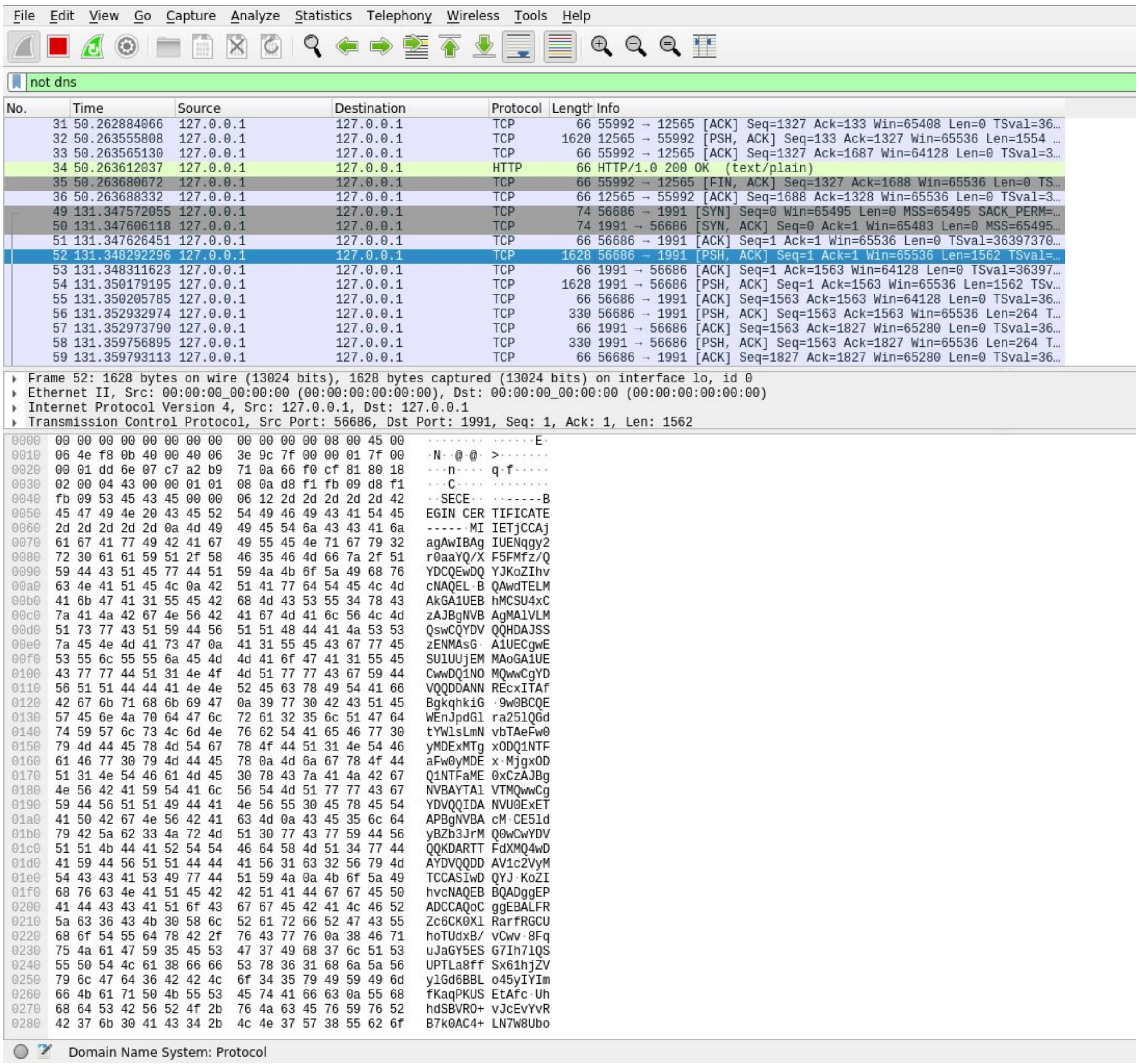


not dns

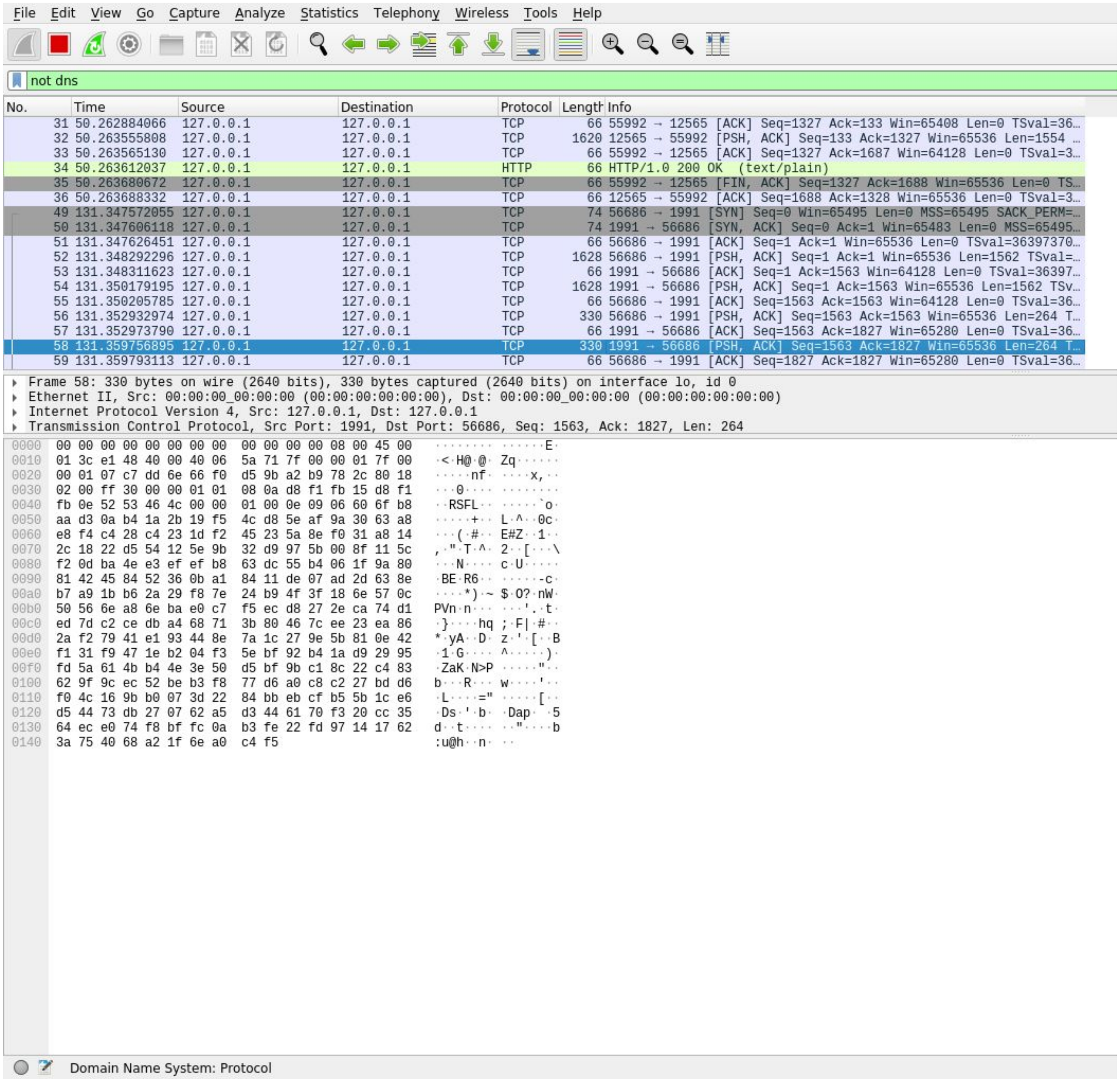
No.	Time	Source	Destination	Protocol	Length	Info
9	38.604211768	127.0.0.1	127.0.0.1	TCP	74	55990 → 12565 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM…
10	38.604236453	127.0.0.1	127.0.0.1	TCP	74	12565 → 55990 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=6549…
11	38.604249603	127.0.0.1	127.0.0.1	TCP	66	55990 → 12565 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=3639644…
12	38.604285970	127.0.0.1	127.0.0.1	TCP	241	55990 → 12565 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=175 TSval=…
13	38.604290603	127.0.0.1	127.0.0.1	TCP	66	12565 → 55990 [ACK] Seq=1 Ack=176 Win=65408 Len=0 TSval=36396…
14	38.604305898	127.0.0.1	127.0.0.1	HTTP	1217	POST /verify HTTP/1.1
15	38.604309486	127.0.0.1	127.0.0.1	TCP	66	12565 → 55990 [ACK] Seq=1 Ack=1327 Win=64384 Len=0 TSval=3639…
16	38.690805293	127.0.0.1	127.0.0.1	TCP	198	12565 → 55990 [PSH, ACK] Seq=1 Ack=1327 Win=65536 Len=132 TSv…
17	38.690822521	127.0.0.1	127.0.0.1	TCP	66	55990 → 12565 [ACK] Seq=1327 Ack=133 Win=65408 Len=0 TSval=36…
18	38.690854938	127.0.0.1	127.0.0.1	TCP	1620	12565 → 55990 [PSH, ACK] Seq=133 Ack=1327 Win=65536 Len=1554 …
19	38.690859732	127.0.0.1	127.0.0.1	TCP	66	55990 → 12565 [ACK] Seq=1327 Ack=1687 Win=64128 Len=0 TSval=3…
20	38.690908130	127.0.0.1	127.0.0.1	HTTP	66	HTTP/1.0 200 OK (text/plain)
21	38.691367826	127.0.0.1	127.0.0.1	TCP	66	55990 → 12565 [FIN, ACK] Seq=1327 Ack=1688 Win=65536 Len=0 TS…
22	38.691377339	127.0.0.1	127.0.0.1	TCP	66	12565 → 55990 [ACK] Seq=1688 Ack=1328 Win=65536 Len=0 TSval=3…
23	50.201028927	127.0.0.1	127.0.0.1	TCP	74	55992 → 12565 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM…
24	50.201042646	127.0.0.1	127.0.0.1	TCP	74	12565 → 55992 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=6549…
25	50.201051171	127.0.0.1	127.0.0.1	TCP	66	55992 → 12565 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=3639655…
26	50.201076996	127.0.0.1	127.0.0.1	TCP	241	55992 → 12565 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=175 TSval=…
27	50.201080517	127.0.0.1	127.0.0.1	TCP	66	12565 → 55992 [ACK] Seq=1 Ack=176 Win=65408 Len=0 TSval=36396…
28	50.201090542	127.0.0.1	127.0.0.1	HTTP	1217	POST /verify HTTP/1.1
29	50.201093125	127.0.0.1	127.0.0.1	TCP	66	12565 → 55992 [ACK] Seq=1 Ack=1327 Win=64384 Len=0 TSval=3639…
30	50.262868594	127.0.0.1	127.0.0.1	TCP	198	12565 → 55992 [PSH, ACK] Seq=1 Ack=1327 Win=65536 Len=132 TSv…
31	50.262884066	127.0.0.1	127.0.0.1	TCP	66	55992 → 12565 [ACK] Seq=1327 Ack=133 Win=65408 Len=0 TSval=36…
32	50.263555808	127.0.0.1	127.0.0.1	TCP	1620	12565 → 55992 [PSH, ACK] Seq=133 Ack=1327 Win=65536 Len=1554 …
33	50.263565130	127.0.0.1	127.0.0.1	TCP	66	55992 → 12565 [ACK] Seq=1327 Ack=1687 Win=64128 Len=0 TSval=3…
34	50.263612037	127.0.0.1	127.0.0.1	HTTP	66	HTTP/1.0 200 OK (text/plain)
35	50.263680672	127.0.0.1	127.0.0.1	TCP	66	55992 → 12565 [FIN, ACK] Seq=1327 Ack=1688 Win=65536 Len=0 TS…
36	50.263688332	127.0.0.1	127.0.0.1	TCP	66	12565 → 55992 [ACK] Seq=1688 Ack=1328 Win=65536 Len=0 TSval=3…
49	131.347572055	127.0.0.1	127.0.0.1	TCP	74	56686 → 1991 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=…
50	131.347606118	127.0.0.1	127.0.0.1	TCP	74	1991 → 56686 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495…
51	131.347626451	127.0.0.1	127.0.0.1	TCP	66	56686 → 1991 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=36397370…
52	131.348292296	127.0.0.1	127.0.0.1	TCP	1628	56686 → 1991 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=1562 TSval=…
53	131.348311623	127.0.0.1	127.0.0.1	TCP	66	1991 → 56686 [ACK] Seq=1 Ack=1563 Win=64128 Len=0 TSval=36397…
54	131.350179195	127.0.0.1	127.0.0.1	TCP	1628	1991 → 56686 [PSH, ACK] Seq=1 Ack=1563 Win=65536 Len=1562 TSv…
55	131.350205785	127.0.0.1	127.0.0.1	TCP	66	56686 → 1991 [ACK] Seq=1563 Ack=1563 Win=64128 Len=0 TSval=36…
56	131.352932974	127.0.0.1	127.0.0.1	TCP	330	56686 → 1991 [PSH, ACK] Seq=1563 Ack=1563 Win=65536 Len=264 T…
57	131.352973790	127.0.0.1	127.0.0.1	TCP	66	1991 → 56686 [ACK] Seq=1563 Ack=1827 Win=65280 Len=0 TSval=36…
58	131.359756895	127.0.0.1	127.0.0.1	TCP	330	1991 → 56686 [PSH, ACK] Seq=1563 Ack=1827 Win=65536 Len=264 T…
59	131.359793113	127.0.0.1	127.0.0.1	TCP	66	56686 → 1991 [ACK] Seq=1827 Ack=1827 Win=65280 Len=0 TSval=36…
60	131.360011036	127.0.0.1	127.0.0.1	TCP	66	1991 → 56686 [FIN, ACK] Seq=1827 Ack=1827 Win=65536 Len=0 TSv…
61	131.365946904	127.0.0.1	127.0.0.1	TCP	66	56686 → 1991 [FIN, ACK] Seq=1827 Ack=1828 Win=65536 Len=0 TSv…
62	131.365982714	127.0.0.1	127.0.0.1	TCP	66	1991 → 56686 [ACK] Seq=1828 Ack=1828 Win=65536 Len=0 TSval=36…
327	264.589950572	127.0.0.1	127.0.0.1	TCP	74	59594 → 1990 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=…
328	264.589995572	127.0.0.1	127.0.0.1	TCP	74	1990 → 59594 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495…
329	264.590034845	127.0.0.1	127.0.0.1	TCP	66	59594 → 1990 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=36398703…
330	264.590990246	127.0.0.1	127.0.0.1	TCP	1628	59594 → 1990 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=1562 TSval=…
331	264.591017420	127.0.0.1	127.0.0.1	TCP	66	1990 → 59594 [ACK] Seq=1 Ack=1563 Win=64128 Len=0 TSval=36398…
332	264.59576985	127.0.0.1	127.0.0.1	TCP	1628	1990 → 59594 [PSH, ACK] Seq=1 Ack=1563 Win=65536 Len=1562 TSv…
333	264.595810190	127.0.0.1	127.0.0.1	TCP	66	59594 → 1990 [ACK] Seq=1563 Ack=1563 Win=64128 Len=0 TSval=36…
334	264.598920864	127.0.0.1	127.0.0.1	TCP	330	59594 → 1990 [PSH, ACK] Seq=1563 Ack=1563 Win=65536 Len=264 T…
335	264.598967731	127.0.0.1	127.0.0.1	TCP	66	1990 → 59594 [ACK] Seq=1563 Ack=1827 Win=65280 Len=0 TSval=36…
336	264.605043835	127.0.0.1	127.0.0.1	TCP	330	1990 → 59594 [PSH, ACK] Seq=1563 Ack=1827 Win=65536 Len=264 T…
337	264.605078270	127.0.0.1	127.0.0.1	TCP	66	59594 → 1990 [ACK] Seq=1827 Ack=1827 Win=65280 Len=0 TSval=36…
338	264.605402408	127.0.0.1	127.0.0.1	TCP	330	1990 → 59594 [PSH, ACK] Seq=1827 Ack=1827 Win=65536 Len=264 T…
339	264.605419917	127.0.0.1	127.0.0.1	TCP	66	59594 → 1990 [ACK] Seq=1827 Ack=2091 Win=65024 Len=0 TSval=36…
340	264.605742955	127.0.0.1	127.0.0.1	TCP	330	1990 → 59594 [PSH, ACK] Seq=2091 Ack=1827 Win=65536 Len=264 T…
341	264.605756750	127.0.0.1	127.0.0.1	TCP	66	59594 → 1990 [ACK] Seq=1827 Ack=2355 Win=64768 Len=0 TSval=36…
342	264.605992226	127.0.0.1	127.0.0.1	TCP	66	1990 → 59594 [FIN, ACK] Seq=2355 Ack=1827 Win=65536 Len=0 TSv…
343	264.627517575	127.0.0.1	127.0.0.1	TCP	66	59594 → 1990 [FIN, ACK] Seq=1827 Ack=2356 Win=65536 Len=0 TSv…
344	264.627561873	127.0.0.1	127.0.0.1	TCP	66	1990 → 59594 [ACK] Seq=2356 Ack=1828 Win=65536 Len=0 TSval=36…

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.


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
not dns
No. Time Source Destination Protocol Length Info
31 50.262884066 127.0.0.1 127.0.0.1 TCP 66 55992 → 12565 [ACK] Seq=1327 Ack=133 Win=65408 Len=0 TSval=36…
32 50.263555808 127.0.0.1 127.0.0.1 TCP 1620 12565 → 55992 [PSH, ACK] Seq=133 Ack=1327 Win=65536 Len=1554 …
33 50.263565130 127.0.0.1 127.0.0.1 TCP 66 55992 → 12565 [ACK] Seq=1327 Ack=1687 Win=64128 Len=0 TSval=3…
34 50.263612037 127.0.0.1 127.0.0.1 HTTP 66 HTTP/1.0 200 OK (text/plain)
35 50.263680672 127.0.0.1 127.0.0.1 TCP 66 55992 → 12565 [FIN, ACK] Seq=1327 Ack=1688 Win=65536 Len=0 TS…
36 50.263688332 127.0.0.1 127.0.0.1 TCP 66 12565 → 55992 [ACK] Seq=1688 Ack=1328 Win=65536 Len=0 TSval=3…
49 131.347572055 127.0.0.1 127.0.0.1 TCP 74 56686 → 1991 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=…
50 131.347606118 127.0.0.1 127.0.0.1 TCP 74 1991 → 56686 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495…
51 131.347626451 127.0.0.1 127.0.0.1 TCP 66 56686 → 1991 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=36397370…
52 131.348292296 127.0.0.1 127.0.0.1 TCP 1628 56686 → 1991 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=1562 TSval=…
53 131.348311623 127.0.0.1 127.0.0.1 TCP 66 1991 → 56686 [ACK] Seq=1 Ack=1563 Win=64128 Len=0 TSval=36397…
54 131.350179195 127.0.0.1 127.0.0.1 TCP 1628 1991 → 56686 [PSH, ACK] Seq=1 Ack=1563 Win=65536 Len=1562 TSv…
55 131.350205785 127.0.0.1 127.0.0.1 TCP 66 56686 → 1991 [ACK] Seq=1563 Ack=1563 Win=64128 Len=0 TSval=36…
56 131.352932974 127.0.0.1 127.0.0.1 TCP 330 56686 → 1991 [PSH, ACK] Seq=1563 Ack=1563 Win=65536 Len=264 T…
57 131.352973790 127.0.0.1 127.0.0.1 TCP 66 1991 → 56686 [ACK] Seq=1563 Ack=1827 Win=65280 Len=0 TSval=36…
58 131.359756895 127.0.0.1 127.0.0.1 TCP 330 1991 → 56686 [PSH, ACK] Seq=1563 Ack=1827 Win=65536 Len=264 T…
59 131.359793113 127.0.0.1 127.0.0.1 TCP 66 56686 → 1991 [ACK] Seq=1827 Ack=1827 Win=65280 Len=0 TSval=36…
Frame 58: 330 bytes on wire (2640 bits), 330 bytes captured (2640 bits) on interface lo, id 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 1991, Dst Port: 56686, Seq: 1563, Ack: 1827, Len: 264
0000 00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00
0010 01 3c e1 48 40 00 40 06 5a 71 7f 00 00 01 7f 00
0020 00 01 07 c7 dd 6e 66 f0 d5 9b a2 b9 78 2c 80 18
0030 02 00 ff 30 00 00 01 01 08 0a d8 f1 fb 15 d8 f1
0040 fb 0e 52 53 46 4c 00 00 01 00 0e 09 06 60 6f b8
0050 aa d3 0a b4 1a 2b 19 f5 4c d8 5e af 9a 30 63 a8
0060 e8 f4 c4 28 c4 23 1d f2 45 23 5a 8e f0 31 a8 14
0070 2c 18 22 d5 54 12 5e 9b 32 d9 97 5b 00 8f 11 5c
0080 f2 0d ba 4e e3 ef ef b8 63 dc 55 b4 06 1f 9a 80
0090 81 42 45 84 52 36 0b a1 84 11 de 07 ad 2d 63 8e
00a0 b7 a9 1b b6 2a 29 f8 7e 24 b9 4f 3f 18 6e 57 0c
00b0 50 56 6e a8 6e ba e0 c7 f5 ec d8 27 2e ca 74 d1
00c0 ed 7d c2 ce db a4 68 71 3b 80 46 7c ee 23 ea 86
00d0 2a f2 79 41 e1 93 44 8e 7a 1c 27 9e 5b 81 0e 42
00e0 f1 31 f9 47 1e b2 04 f3 5e bf 92 b4 1a d9 29 95
00f0 fd 5a 61 4b b4 4e 3e 50 d5 bf 9b c1 8c 22 c4 83
0100 62 9f 9c ec 52 be b3 f8 77 d6 a0 c8 c2 27 bd d6
0110 f0 4c 16 9b b0 07 3d 22 84 bb eb cf b5 5b 1c e6
0120 d5 44 73 db 27 07 62 a5 d3 44 61 70 f3 20 cc 35
0130 64 ec e0 74 f8 bf fc 0a b3 fe 22 fd 97 14 17 62
0140 3a 75 40 68 a2 1f 6e a0 c4 f5
Domain Name System: Protocol

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.