भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Secure Chat Application

Network Security Project Report

Submitted by

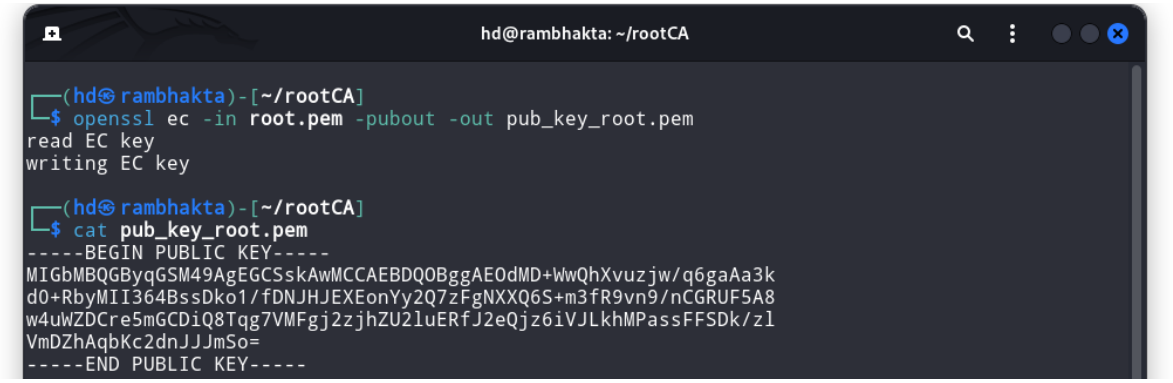
Patel Heetkumar Dilipbhai	CS23MTECH11029
KR Anuraj	CS23MTECH13002
Vishal Patidar	CS23MTECH14017

Task 1 – Keys & Certificate generation

Root CA's Certificate Generation :

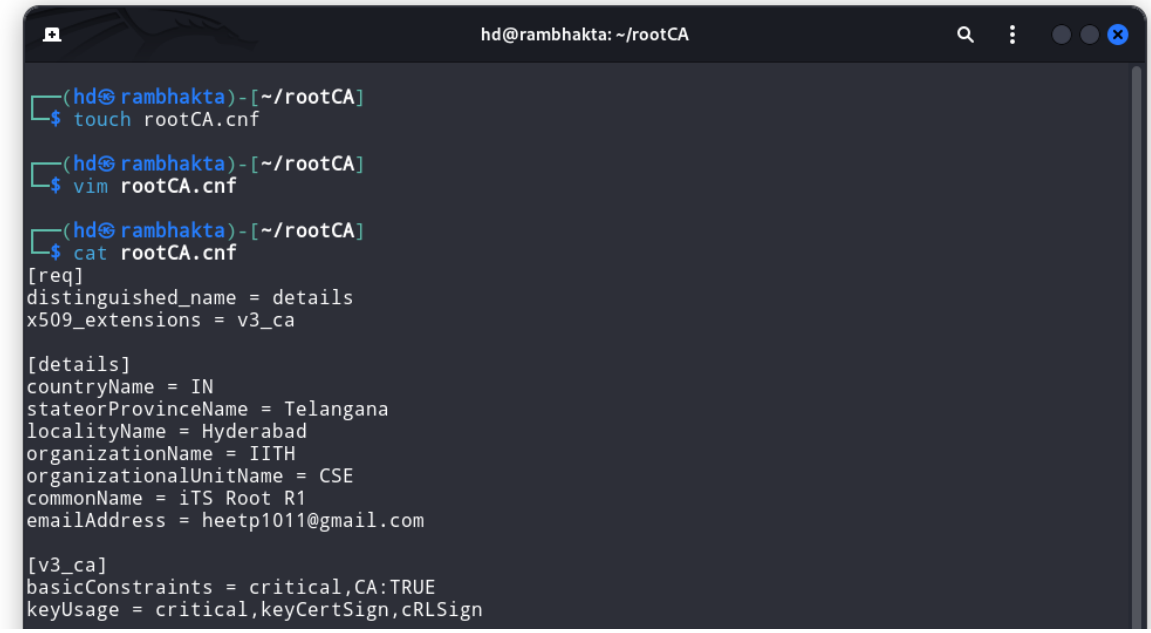
Step 1 : Generating ECC 512-bit Key pair for Root CA

Step 2 : Extracting Public key from the above generated key pair



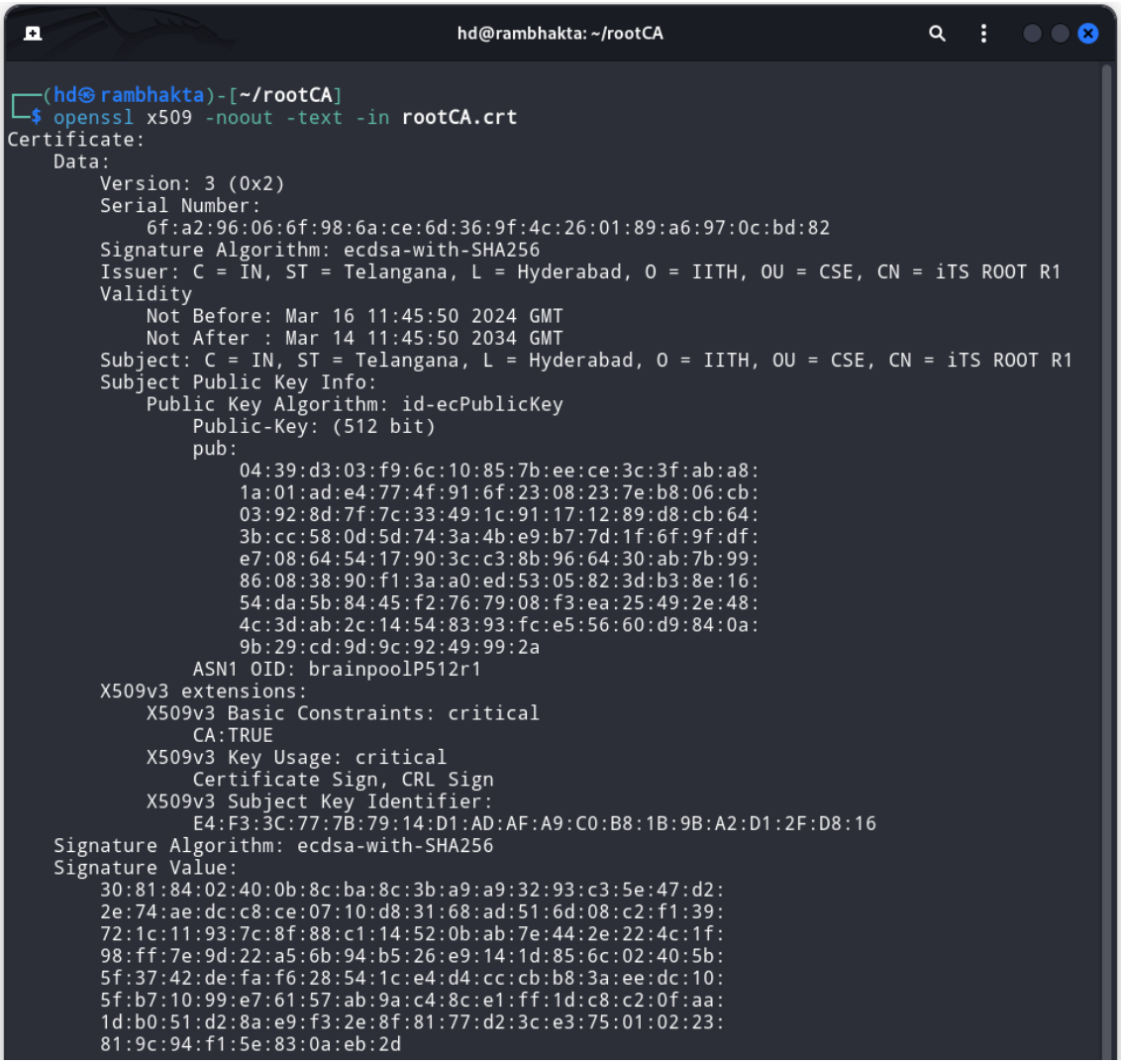
Now in order to generate the self-sign certificate root CA need one config file which contains all details for generating certificate including the basic constraints, key usage and more.

Step 3 : Creating config file for generating self-signed certificate



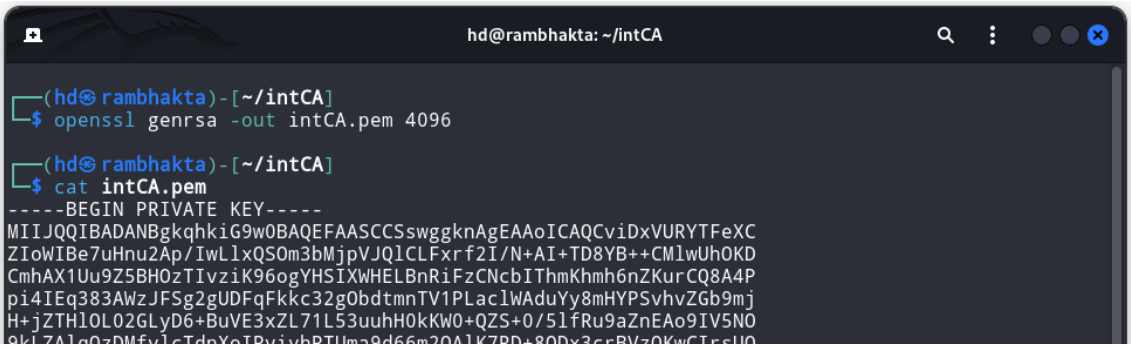
Step 4 : Creating self-signed certificate for root CA

Output : Root CA's self-signed certificate



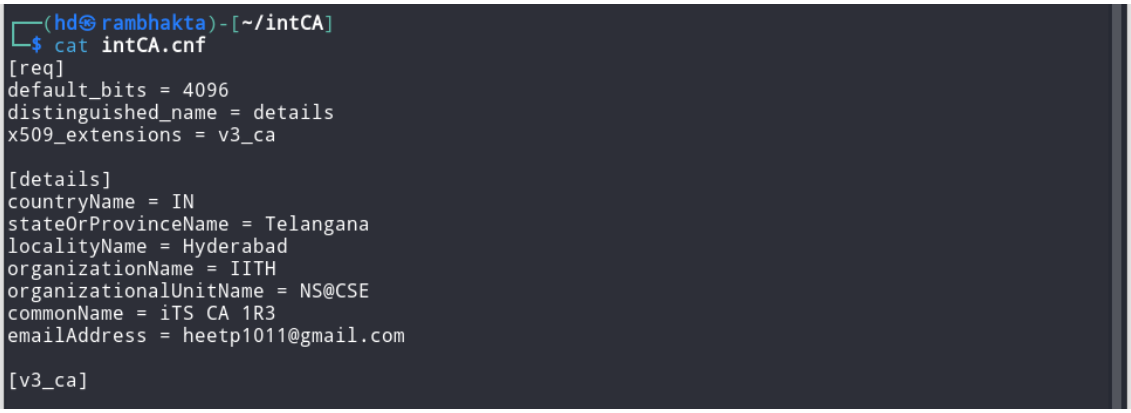
Intermediate CA's Certificate Generation :

Step 1 : Generating 4096-bit RSA Key pair for Int CA

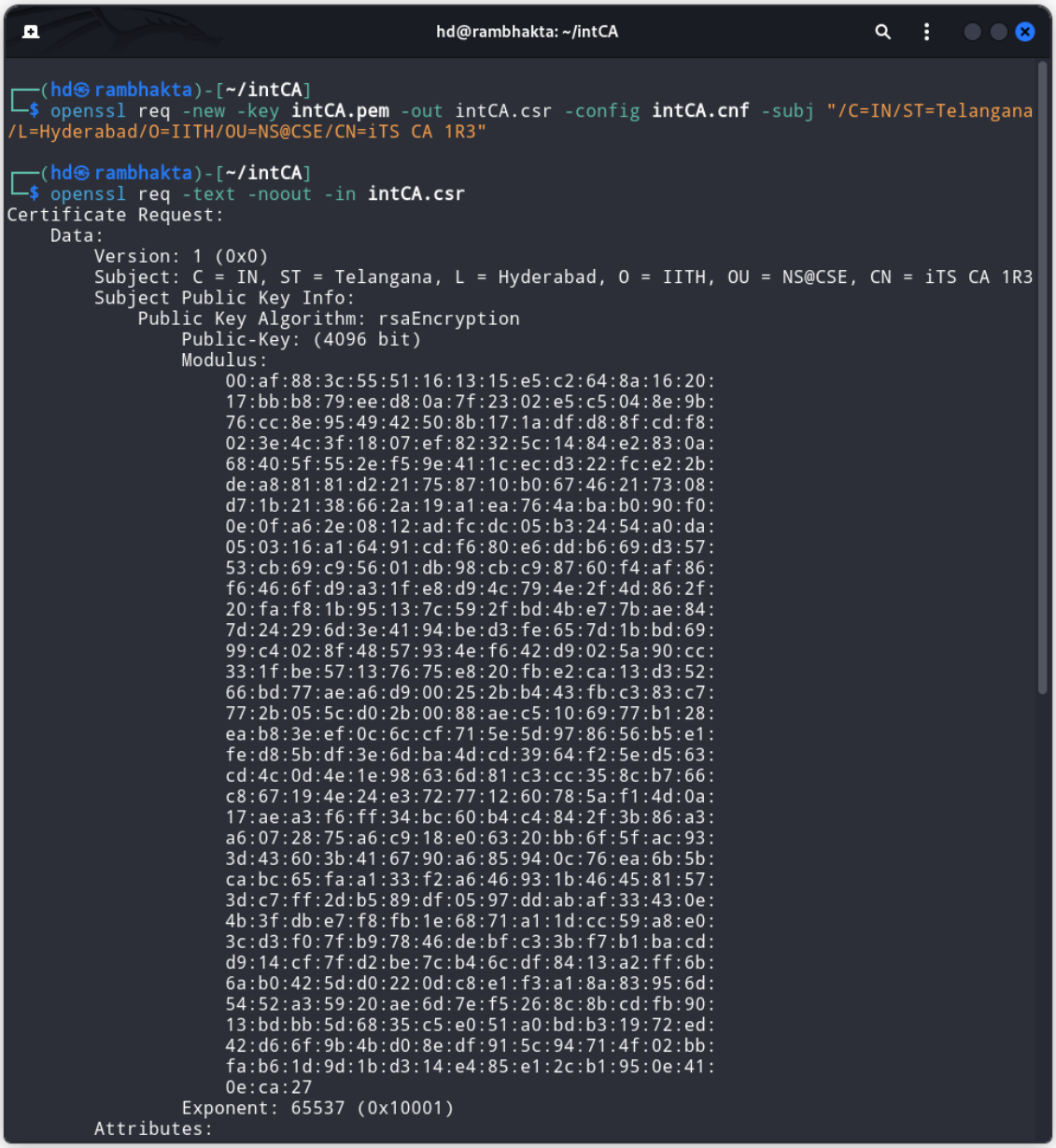


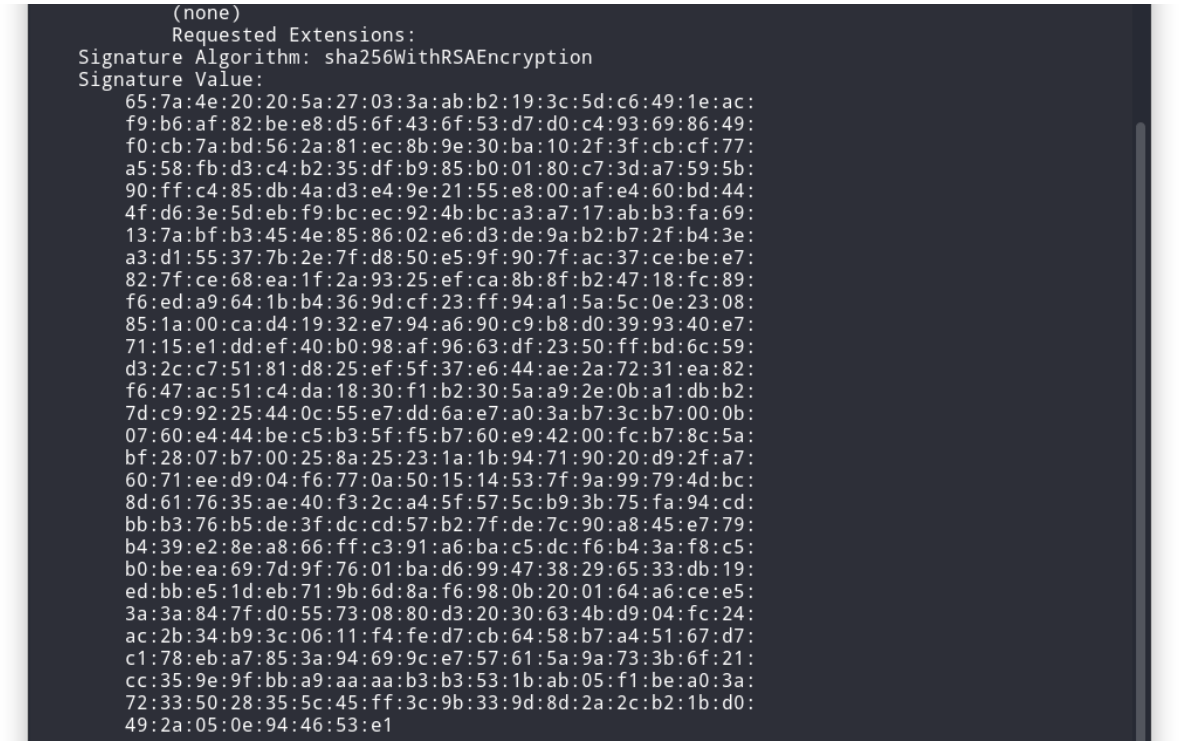
Step 2 : Extracting Public key from the above generated key pair

Step 3 : Creating config file for Int CA to generate certificate signing request

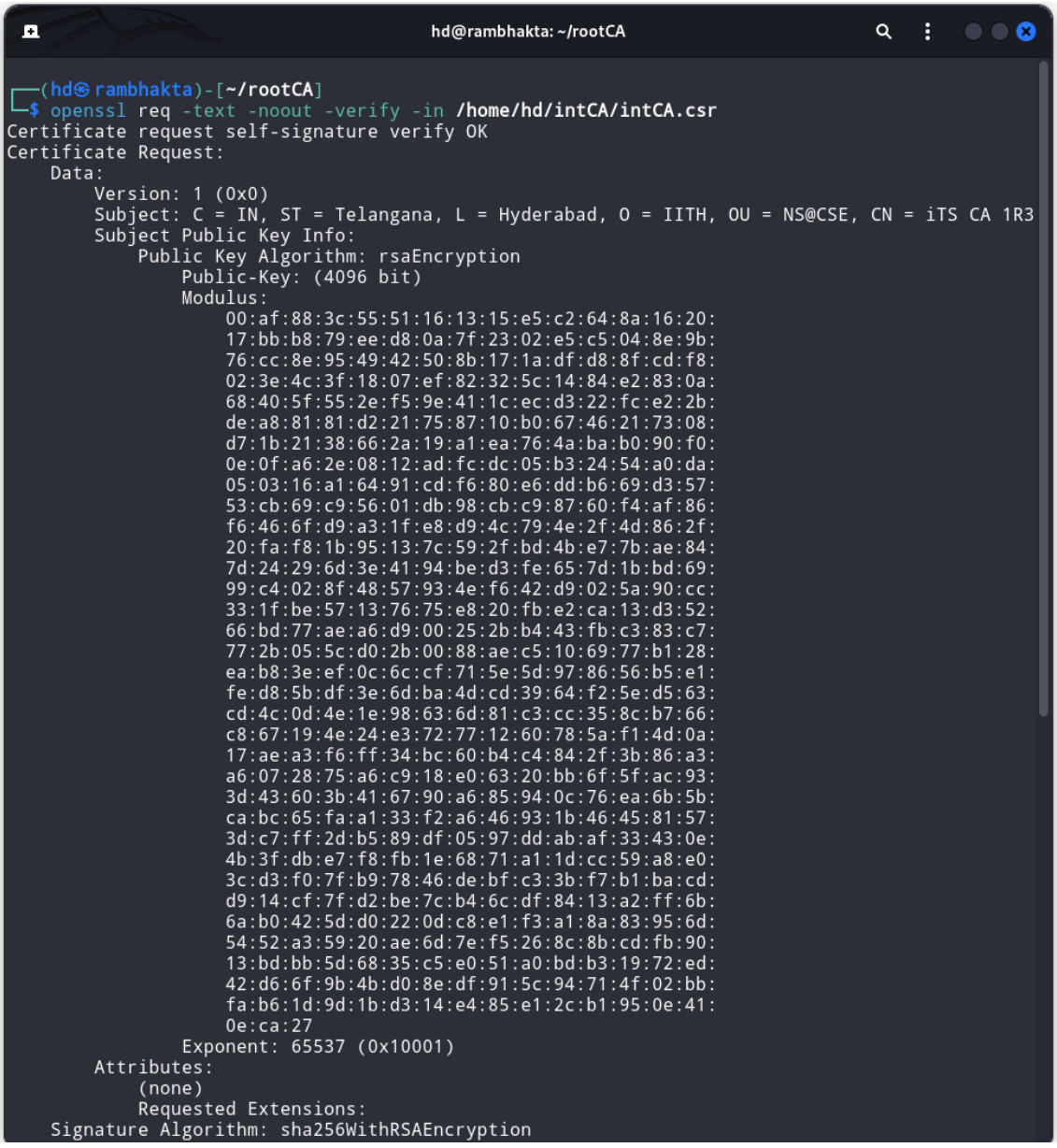


Step 4 : Creating certificate signing request with above created config file and sending it to Root CA



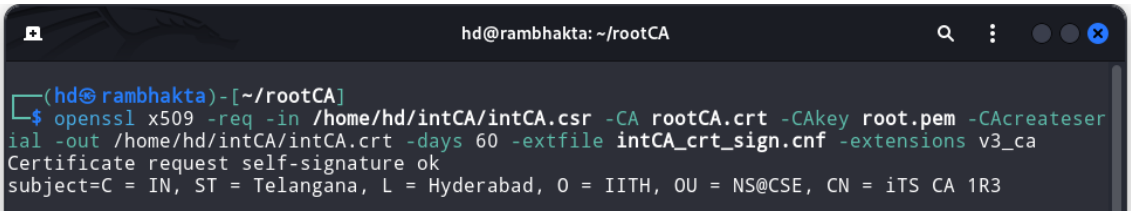

(none)
Requested Extensions:
Signature Algorithm: sha256WithRSAEncryption
Signature Value:
65:7a:4e:20:20:5a:27:03:3a:ab:b2:19:3c:5d:c6:49:1e:ac:
f9:b6:af:82:be:e8:d5:6f:43:6f:53:d7:d0:c4:93:69:86:49:
f0:cb:7a:bd:56:2a:81:ec:8b:9e:30:ba:10:2f:3f:cb:cf:77:
a5:58:fb:d3:c4:b2:35:df:b9:85:b0:01:80:c7:3d:a7:59:5b:
90:ff:c4:85:db:4a:d3:e4:9e:21:55:e8:00:af:e4:60:bd:44:
4f:d6:3e:5d:eb:f9:bc:ec:92:4b:bc:a3:a7:17:ab:b3:fa:69:
13:7a:bf:b3:45:4e:85:86:02:e6:d3:de:9a:b2:b7:2f:b4:3e:
a3:d1:55:37:7b:2e:7f:d8:50:e5:9f:90:7f:ac:37:ce:be:e7:
82:7f:ce:68:ea:1f:2a:93:25:ef:ca:8b:8f:b2:47:18:fc:89:
f6:ed:a9:64:1b:b4:36:9d:cf:23:ff:94:a1:5a:5c:0e:23:08:
85:1a:00:ca:d4:19:32:e7:94:a6:90:c9:b8:d0:39:93:40:e7:
71:15:e1:dd:ef:40:b0:98:af:96:63:df:23:50:ff:bd:6c:59:
d3:2c:c7:51:81:d8:25:ef:5f:37:e6:44:ae:2a:72:31:ea:82:
f6:47:ac:51:c4:da:18:30:f1:b2:30:5a:a9:2e:0b:a1:db:b2:
7d:c9:92:25:44:0c:55:e7:dd:6a:e7:a0:3a:b7:3c:b7:00:0b:
07:60:e4:44:be:c5:b3:5f:f5:b7:60:e9:42:00:fc:b7:8c:5a:
bf:28:07:b7:00:25:8a:25:23:1a:1b:94:71:90:20:d9:2f:a7:
60:71:ee:d9:04:f6:77:0a:50:15:14:53:7f:9a:99:79:4d:bc:
8d:61:76:35:ae:40:f3:2c:a4:5f:57:5c:b9:3b:75:fa:94:cd:
bb:b3:76:b5:de:3f:dc:cd:57:b2:7f:de:7c:90:a8:45:e7:79:
b4:39:e2:8e:a8:66:ff:c3:91:a6:ba:c5:dc:f6:b4:3a:f8:c5:
b0:be:ea:69:7d:9f:76:01:ba:d6:99:47:38:29:65:33:db:19:
ed:bb:e5:1d:eb:71:9b:6d:8a:f6:98:0b:20:01:64:a6:ce:e5:
3a:3a:84:7f:d0:55:73:08:80:d3:20:30:63:4b:d9:04:fc:24:
ac:2b:34:b9:3c:06:11:f4:fe:d7:cb:64:58:b7:a4:51:67:d7:
c1:78:eb:a7:85:3a:94:69:9c:e7:57:61:5a:9a:73:3b:6f:21:
cc:35:9e:9f:bb:a9:aa:aa:b3:b3:53:1b:ab:05:f1:be:a0:3a:
72:33:50:28:35:5c:45:ff:3c:9b:33:9d:8d:2a:2c:b2:1b:d0:
49:2a:05:0e:94:46:53:e1

Step 5 : On receiving csr, Root CA will verify the cse that it is authentic.



Step 6 : Root CA then creates the config file which contains the basic constraints and key usage details.

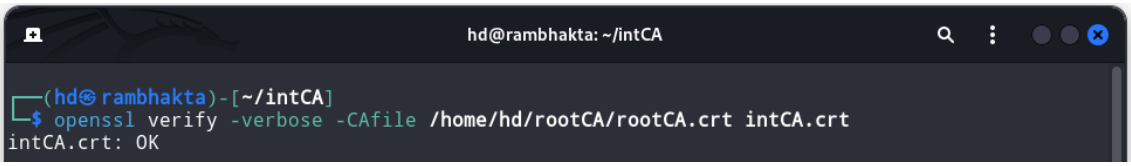
Step 7 : Creating certificate with above generate config file and received csr file.



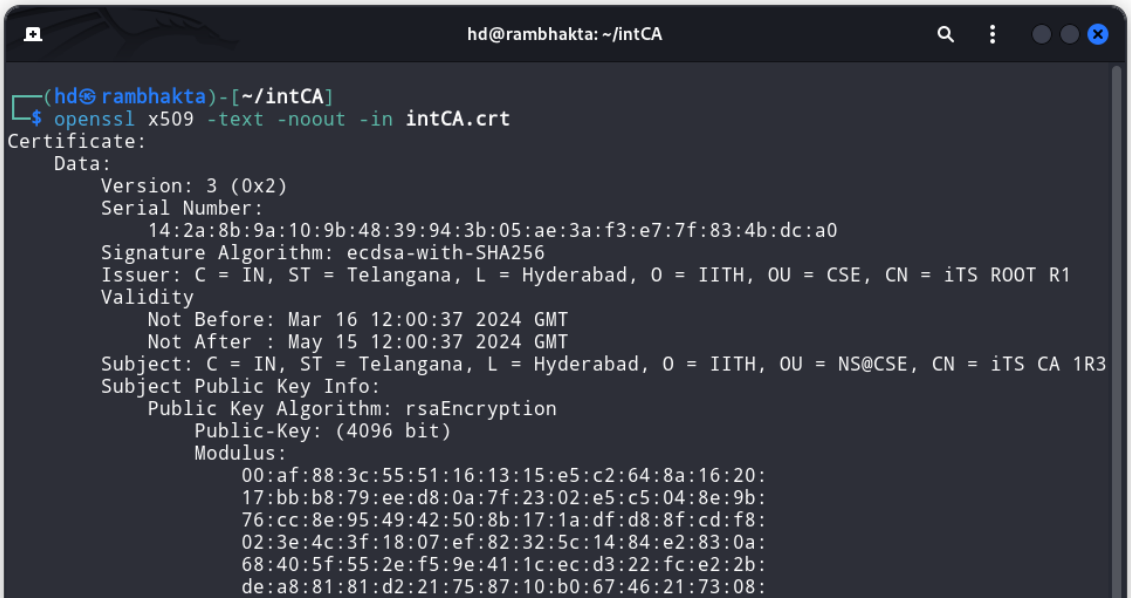
Step 8 : Creating certificate chain with rootCA.crt and IntCA.crt and sending it to IntCA.

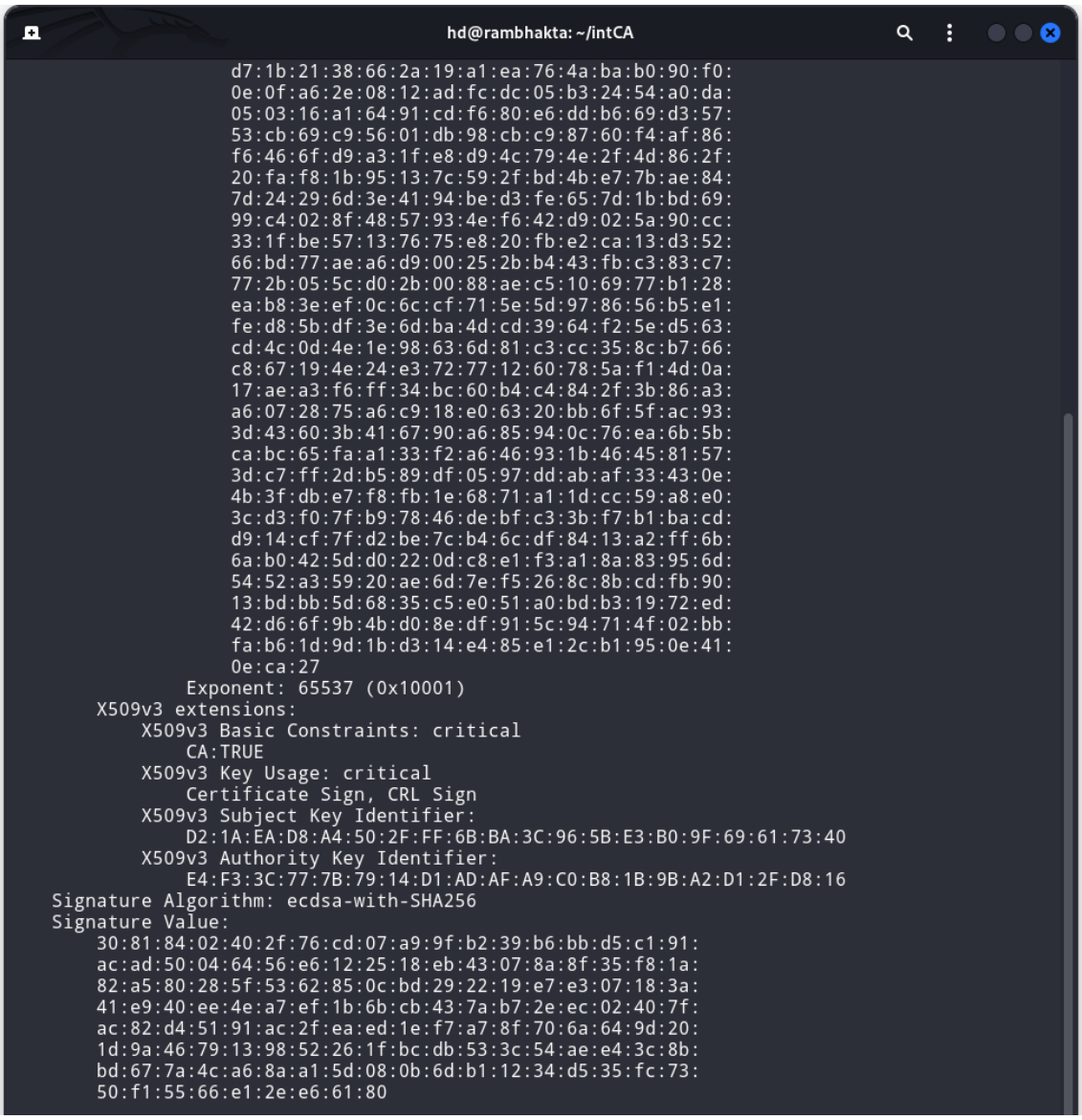
“cat intCA.crt rootCA.crt > /home/hd/intCA/intCA_crt_chain.crt”

Step 9 : Verifying authenticity of the certificate means verifying that it is signed by trusted authority.



Output : Int CA's certificate signed and verified by Root CA.

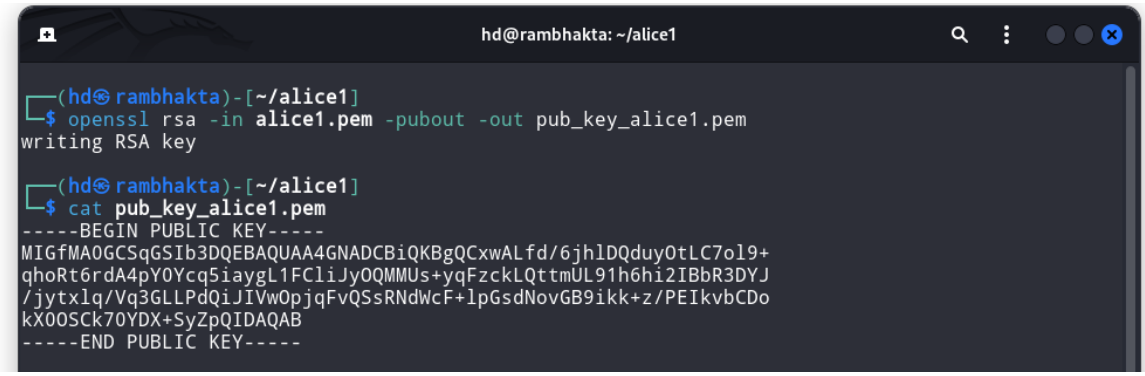




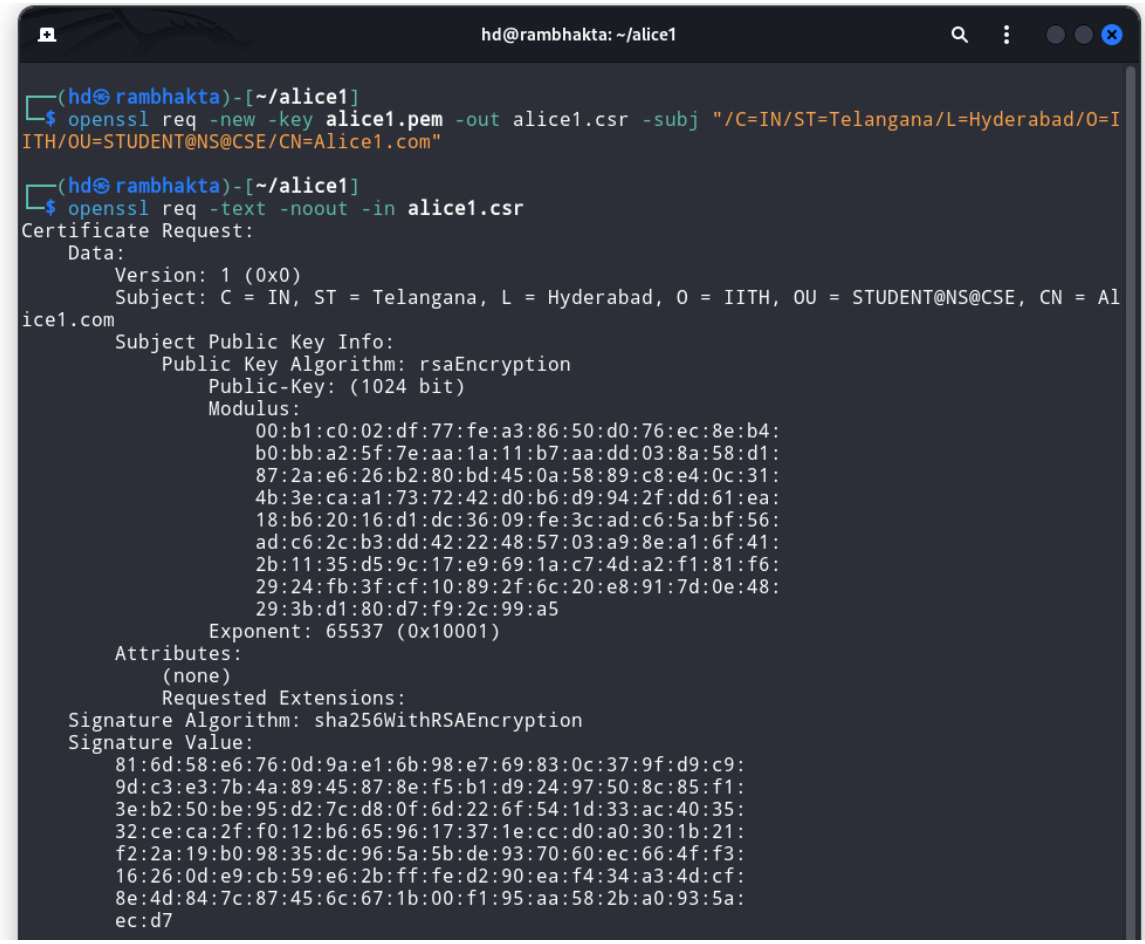
Alice's Certificate Generation :

Step 1 : Generating 1024-bit RSA Key pair for Alice

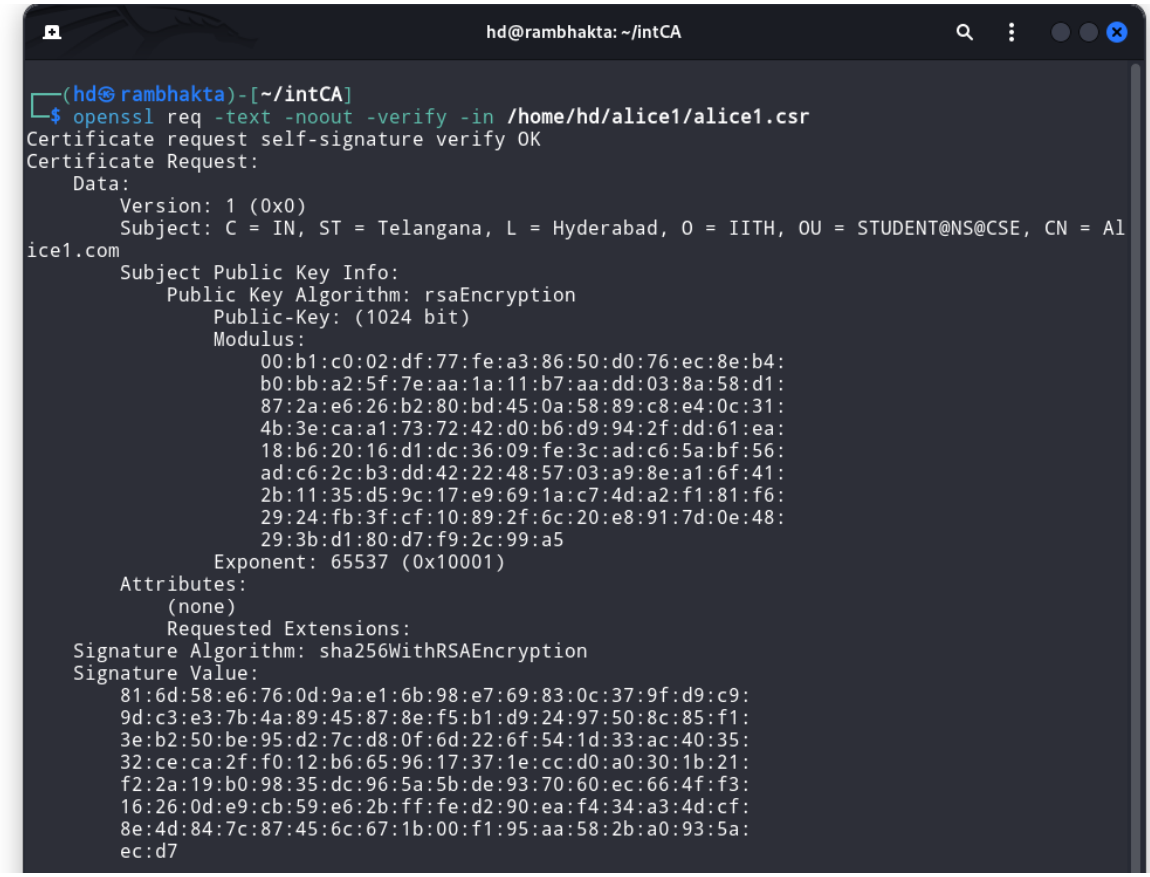
Step 2 : Extracting Public key from the above generated key pair



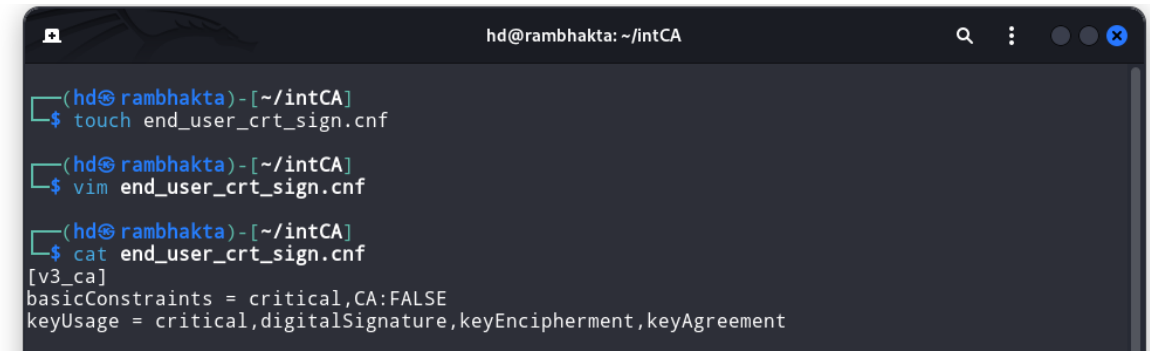
Step 3 : Alice creates it certificate signing request and send it to Int CA



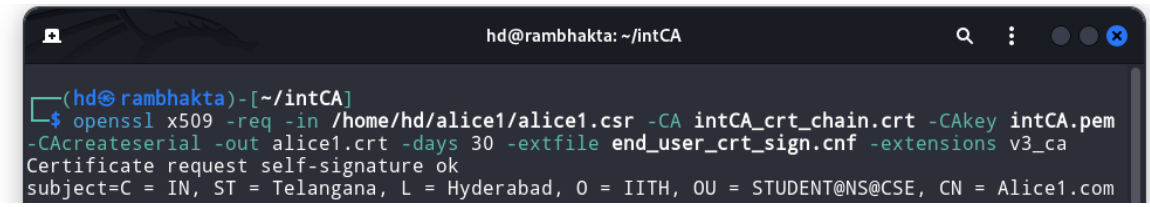
Step 4 : Int CA verifies the received csr from Alice



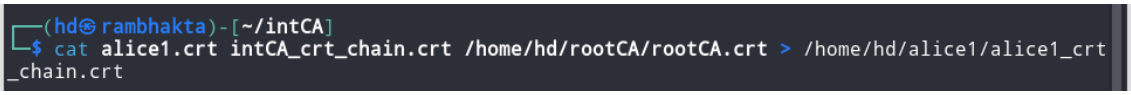
Step 5 : Int CA then creates the config file which contains the basic constraints and key usage details.



Step 6 : Generating certificate for Alice using received csr and above generated config file.

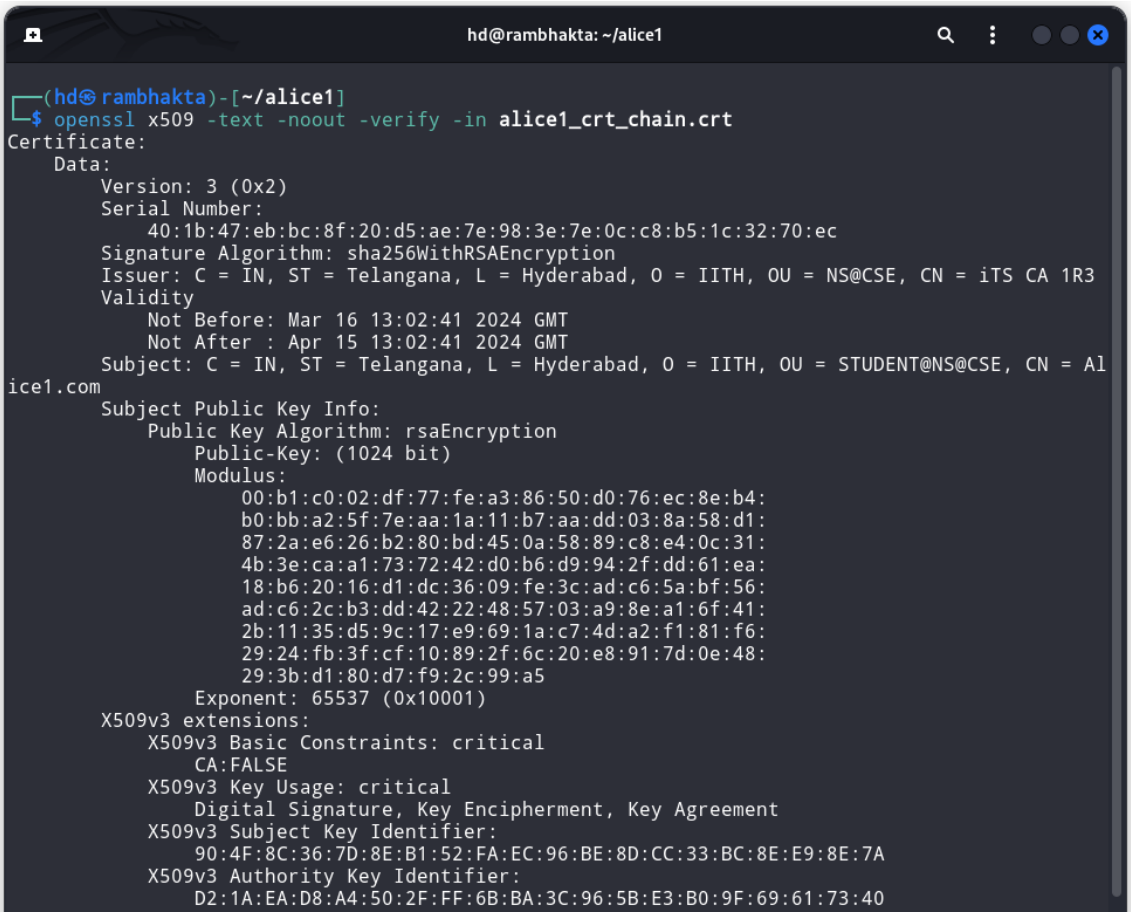


Step 7 : Generating certificate chain by attaching rootCA.crt and intCA.crt and sending to Alice.

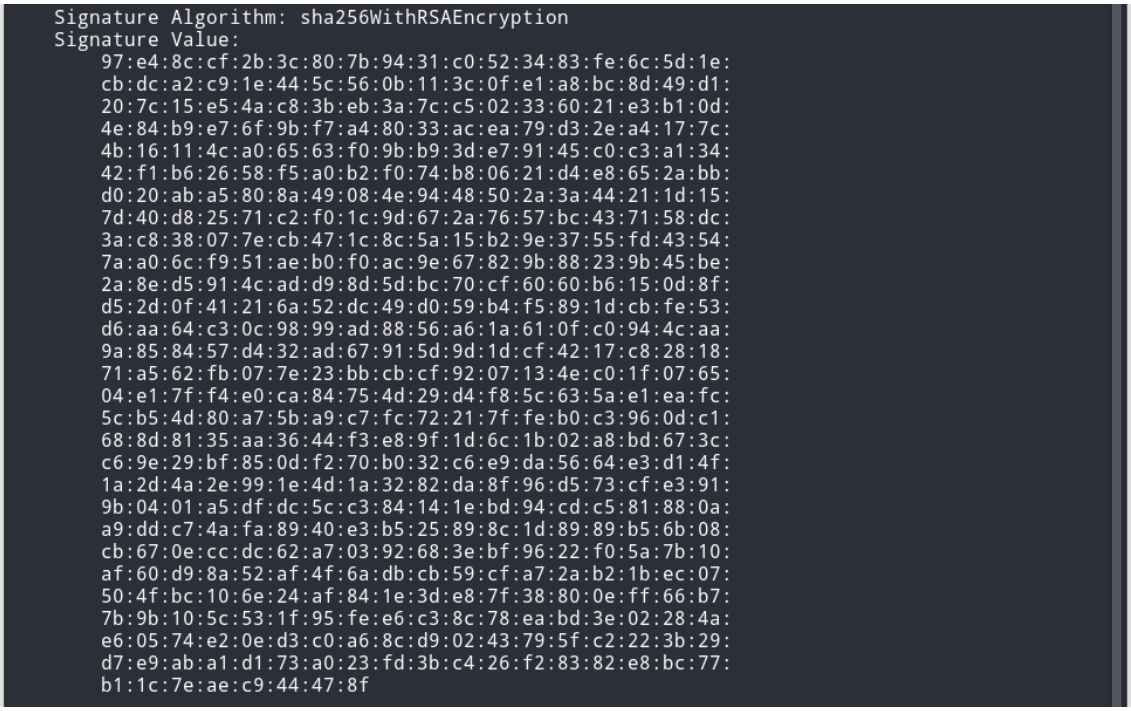


Step 8 : Alice verifies authenticity of the certificate means verifying that it is signed by trusted authority i.e. Int CA.

Output : Alice's certificate chain signed by Int CA.



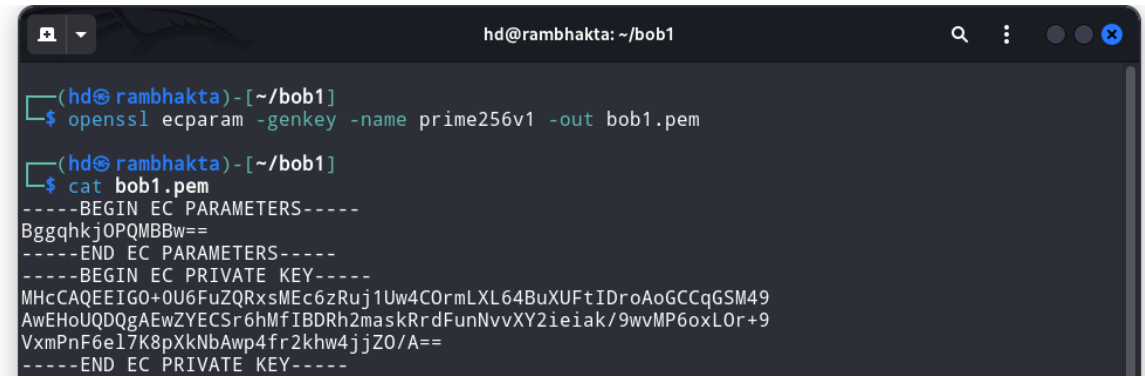
```
hd@rambhakta: ~/alice1

┌──(hd㉿rambhakta)-[~/alice1]
└─$ openssl x509 -text -noout -verify -in alice1_crt_chain.crt
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            40:1b:47:eb:bc:8f:20:d5:ae:7e:98:3e:7e:0c:c8:b5:1c:32:70:ec
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = IN, ST = Telangana, L = Hyderabad, O = IITH, OU = NS@CSE, CN = iTS CA 1R3
        Validity
            Not Before: Mar 16 13:02:41 2024 GMT
            Not After : Apr 15 13:02:41 2024 GMT
        Subject: C = IN, ST = Telangana, L = Hyderabad, O = IITH, OU = STUDENT@NS@CSE, CN = Al
ice1.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (1024 bit)
                Modulus:
                    00:b1:c0:02:df:77:fe:a3:86:50:d0:76:ec:8e:b4:
                    b0:bb:a2:5f:7e:aa:1a:11:b7:aa:dd:03:8a:58:d1:
                    87:2a:e6:26:b2:80:bd:45:0a:58:89:c8:e4:0c:31:
                    4b:3e:ca:a1:73:72:42:d0:b6:d9:94:2f:dd:61:ea:
                    18:b6:20:16:d1:dc:36:09:fe:3c:ad:c6:5a:bf:56:
                    ad:c6:2c:b3:dd:42:22:48:57:03:a9:8e:a1:6f:41:
                    2b:11:35:d5:9c:17:e9:69:1a:c7:4d:a2:f1:81:f6:
                    29:24:fb:3f:cf:10:89:2f:6c:20:e8:91:7d:0e:48:
                    29:3b:d1:80:d7:f9:2c:99:a5
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment, Key Agreement
            X509v3 Subject Key Identifier:
                90:4F:8C:36:7D:8E:B1:52:FA:EC:96:BE:8D:CC:33:BC:8E:E9:8E:7A
            X509v3 Authority Key Identifier:
                D2:1A:EA:D8:A4:50:2F:FF:6B:BA:3C:96:5B:E3:B0:9F:69:61:73:40
```



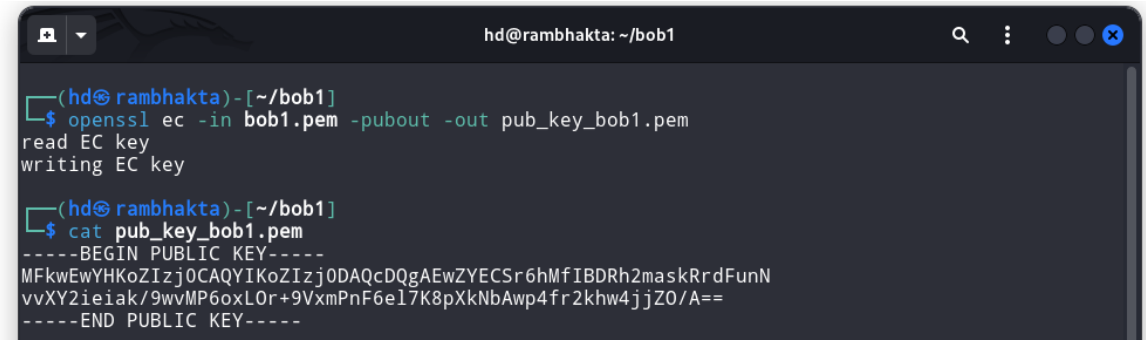
```
    Signature Algorithm: sha256WithRSAEncryption
    Signature Value:
        97:e4:8c:cf:2b:3c:80:7b:94:31:c0:52:34:83:fe:6c:5d:1e:
        cb:dc:a2:c9:1e:44:5c:56:0b:11:3c:0f:e1:a8:bc:8d:49:d1:
        20:7c:15:e5:4a:c8:3b:eb:3a:7c:c5:02:33:60:21:e3:b1:0d:
        4e:84:b9:e7:6f:9b:f7:a4:80:33:ac:ea:79:d3:2e:a4:17:7c:
        4b:16:11:4c:a0:65:63:f0:9b:b9:3d:e7:91:45:c0:c3:a1:34:
        42:f1:b6:26:58:f5:a0:b2:f0:74:b8:06:21:d4:e8:65:2a:bb:
        d0:20:ab:a5:80:8a:49:08:4e:94:48:50:2a:3a:44:21:1d:15:
        7d:40:d8:25:71:c2:f0:1c:9d:67:2a:76:57:bc:43:71:58:dc:
        3a:c8:38:07:7e:cb:47:1c:8c:5a:15:b2:9e:37:55:fd:43:54:
        7a:a0:6c:f9:51:ae:b0:f0:ac:9e:67:82:9b:88:23:9b:45:be:
        2a:8e:d5:91:4c:ad:d9:8d:5d:bc:70:cf:60:60:b6:15:0d:8f:
        d5:2d:0f:41:21:6a:52:dc:49:d0:59:b4:f5:89:1d:cb:fe:53:
        d6:aa:64:c3:0c:98:99:ad:88:56:a6:1a:61:0f:c0:94:4c:aa:
        9a:85:84:57:d4:32:ad:67:91:5d:9d:1d:cf:42:17:c8:28:18:
        71:a5:62:fb:07:7e:23:bb:cb:cf:92:07:13:4e:c0:1f:07:65:
        04:e1:7f:f4:e0:ca:84:75:4d:29:d4:f8:5c:63:5a:e1:ea:fc:
        5c:b5:4d:80:a7:5b:a9:c7:fc:72:21:7f:fe:b0:c3:96:0d:c1:
        68:8d:81:35:aa:36:44:f3:e8:9f:1d:6c:1b:02:a8:bd:67:3c:
        c6:9e:29:bf:85:0d:f2:70:b0:32:c6:e9:da:56:64:e3:d1:4f:
        1a:2d:4a:2e:99:1e:4d:1a:32:82:da:8f:96:d5:73:cf:e3:91:
        9b:04:01:a5:df:dc:5c:c3:84:14:1e:bd:94:cd:c5:81:88:0a:
        a9:dd:c7:4a:fa:89:40:e3:b5:25:89:8c:1d:89:89:b5:6b:08:
        cb:67:0e:cc:dc:62:a7:03:92:68:3e:bf:96:22:f0:5a:7b:10:
        af:60:d9:8a:52:af:4f:6a:db:cb:59:cf:a7:2a:b2:1b:ec:07:
        50:4f:bc:10:6e:24:af:84:1e:3d:e8:7f:38:80:0e:ff:66:b7:
        7b:9b:10:5c:53:1f:95:fe:e6:c3:8c:78:ea:bd:3e:02:28:4a:
        e6:05:74:e2:0e:d3:c0:a6:8c:d9:02:43:79:5f:c2:22:3b:29:
        d7:e9:ab:a1:d1:73:a0:23:fd:3b:c4:26:f2:83:82:e8:bc:77:
        b1:1c:7e:ae:c9:44:47:8f
```

Bob’s Certificate Generation :

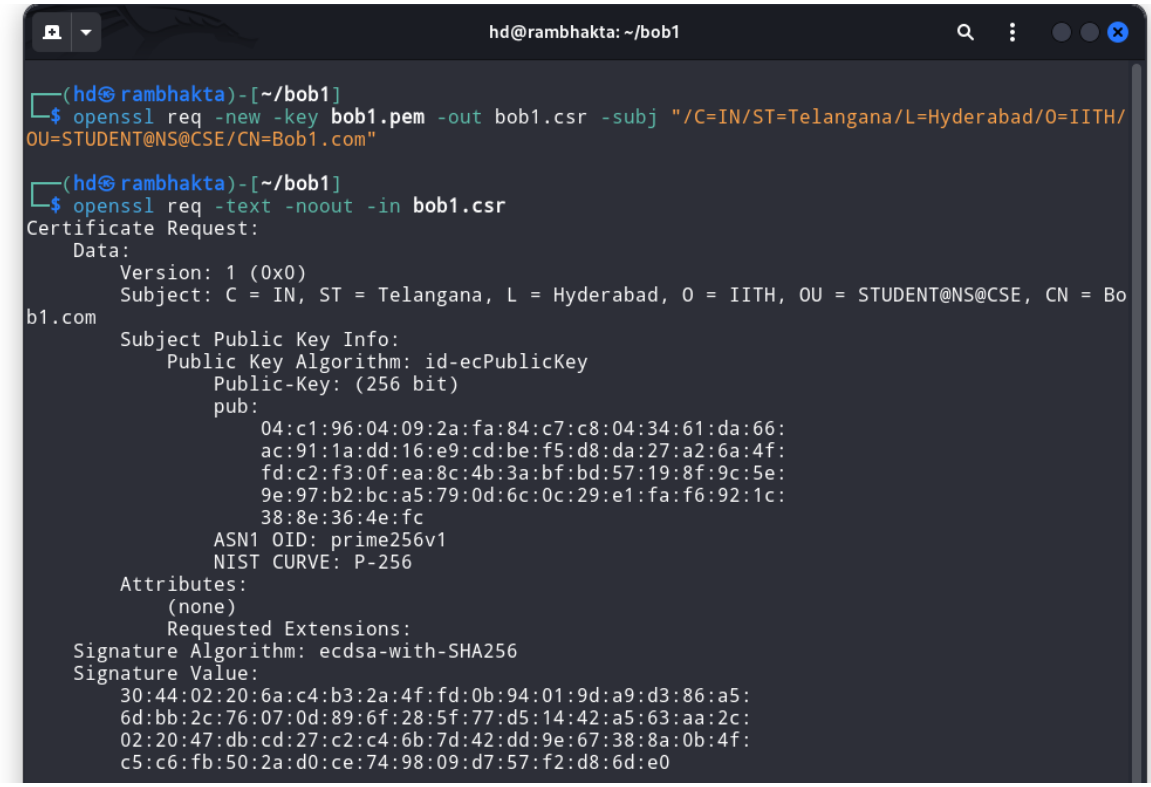
Step 1 : Generating 256-bit ECC Key pair for Bob.



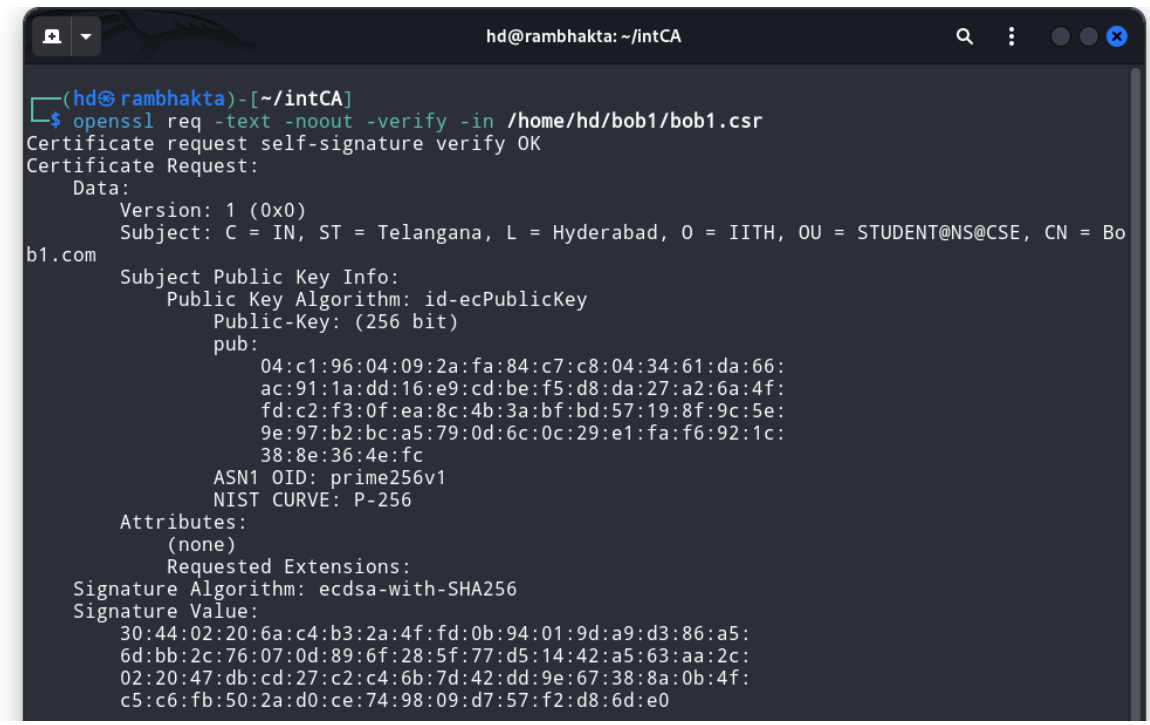
Step 2 : Extracting Public key from the above generated key pair.



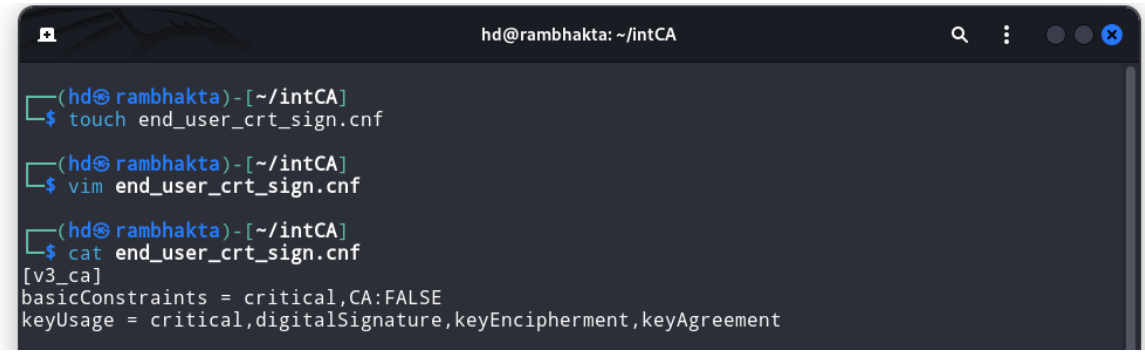
Step 3 : Bob creates it certificate signing request and send it to Int CA.



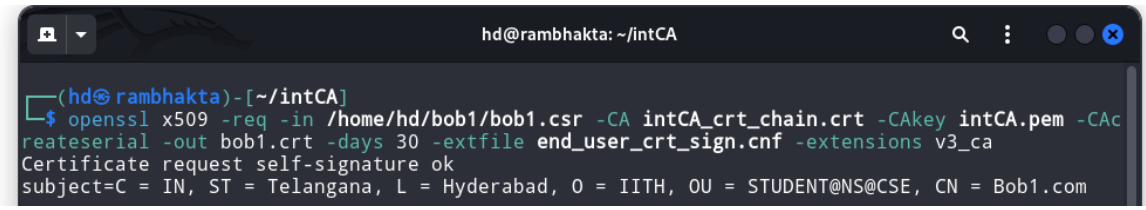
Step 4 : Int CA verifies the received csr from Bob.



Step 5 : Int CA then creates the config file which contains the basic constraints and key usage details.

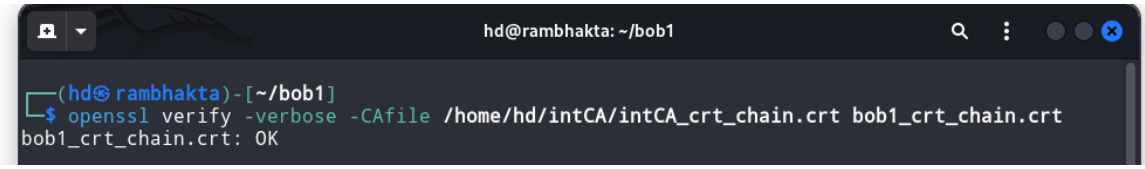


Step 6 : Generating certificate for Bob using received csr and above generated config file.

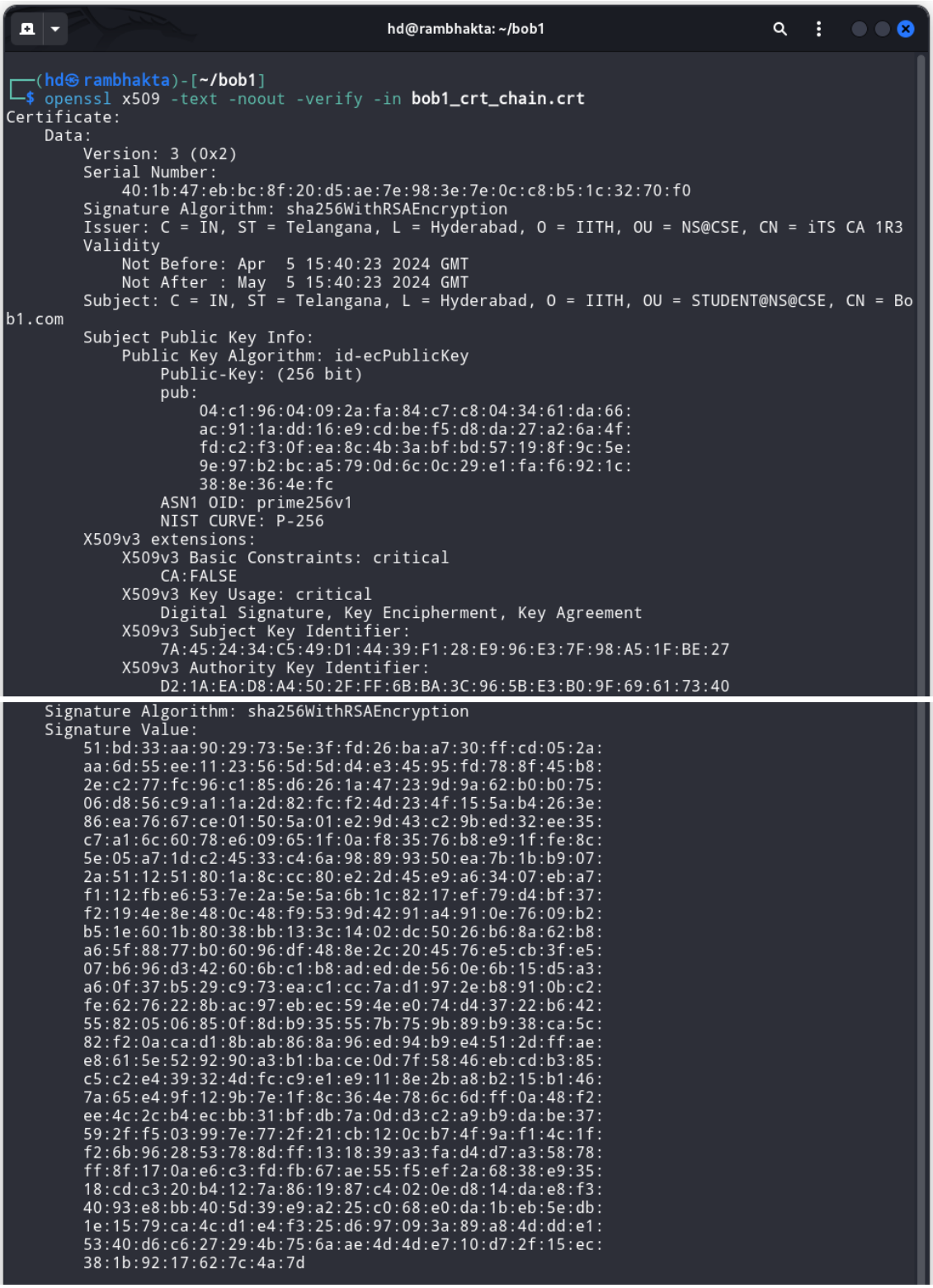


Step 7 : Generating certificate chain by attaching rootCA.crt and intCA.crt and sending to Bob.

Step 8 : Bob verifies authenticity of the certificate means verifying that it is signed by trusted authority i.e. Int CA.

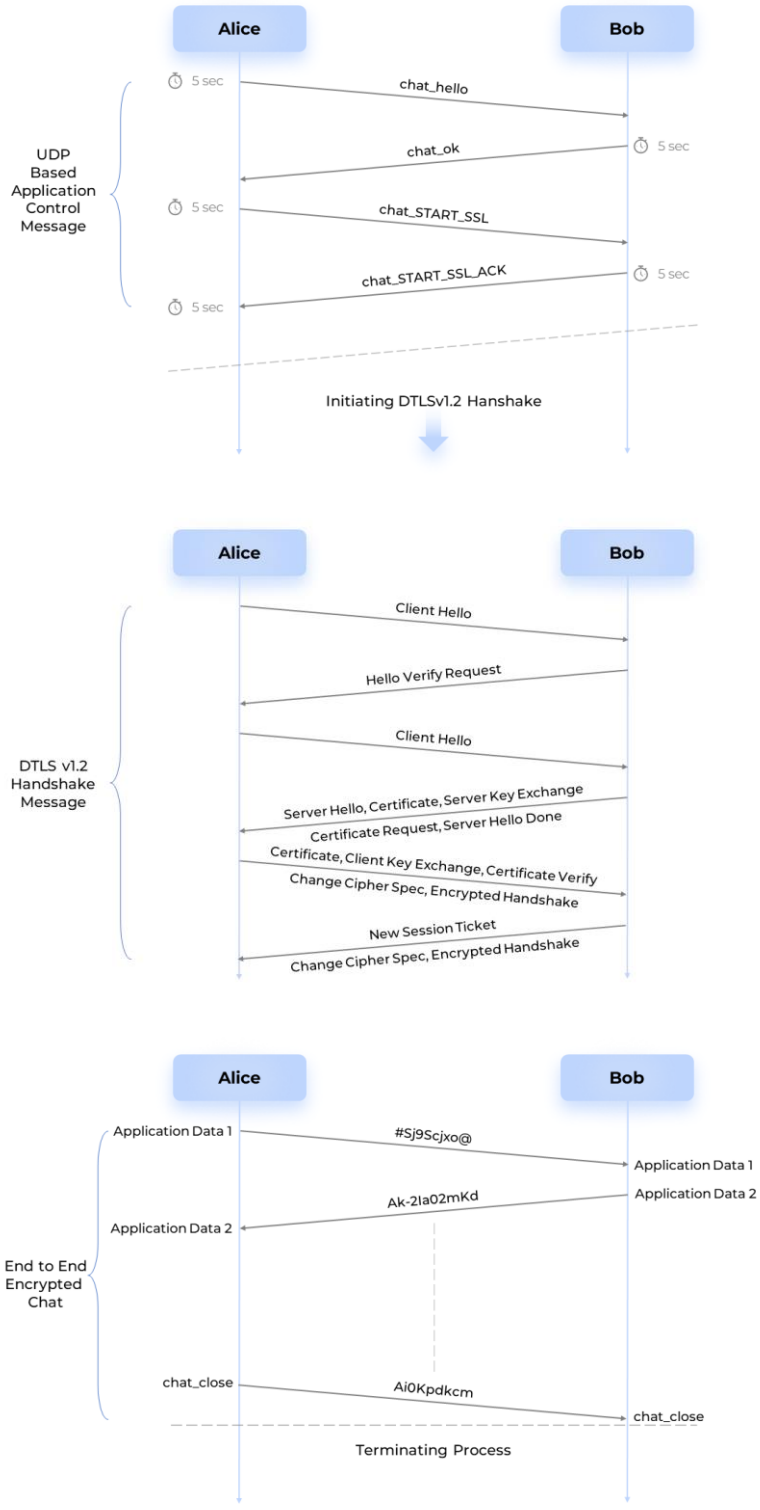


Output : Bob's certificate chain signed by Int CA.



Task 2 – Secure Chat Application

Application Flow :



Above exceptional flow diagram created by one of our team member shows the basic working of our chat application.

Assuming Client & Server both have certificates of Root CA and Int CA preloaded in their respective trust stores.

Some code snippets :

Below code snippet is responsible for loading necessary modules under openssl library, and for creating the SSL context of client & server.

```
28  void initialize_openssl() {
29          SSL_library_init();
30          SSL_load_error_strings();
31          OpenSSL_add_ssl_algorithms();
32          ERR_load_crypto_strings();
33  }
34
35  SSL_CTX* create_server_context() {
36          const SSL_METHOD *method = DTLSv1_2_server_method();
37          SSL_CTX *ctx = SSL_CTX_new(method);
38          if (!ctx) {
39                  perror("Unable to create SSL context");
40                  ERR_print_errors_fp(stderr);
41                  exit(EXIT_FAILURE);
42          }
43          return ctx;
44  }
45
46  SSL_CTX* create_client_context() {
47      const SSL_METHOD* method = DTLSv1_2_client_method();
48      SSL_CTX* ctx = SSL_CTX_new(method);
49      if (!ctx) {
50          std::cerr << "Unable to create SSL context" << std::endl;
51          ERR_print_errors_fp(stderr);
52          exit(EXIT_FAILURE);
53      }
54      return ctx;
55  }
```

Below code snippet helps to set socket in non-blocking mode which here helps as a part of retries and retransmission functionality.

```
105  void setup_nonblocking(){
106          int fcntl_check = fcntl(socket_descriptor, F_GETFL, 0);
107          if (fcntl_check == -1) {
108                  perror("F_GETFL error\n");
109                  exit(1);
110          }
111
112          if (fcntl(socket_descriptor, F_SETFL, fcntl_check | O_NONBLOCK) == -1) {
113                  perror("F_SETFL error\n");
114                  exit(1);
115          }
116  }
117
```

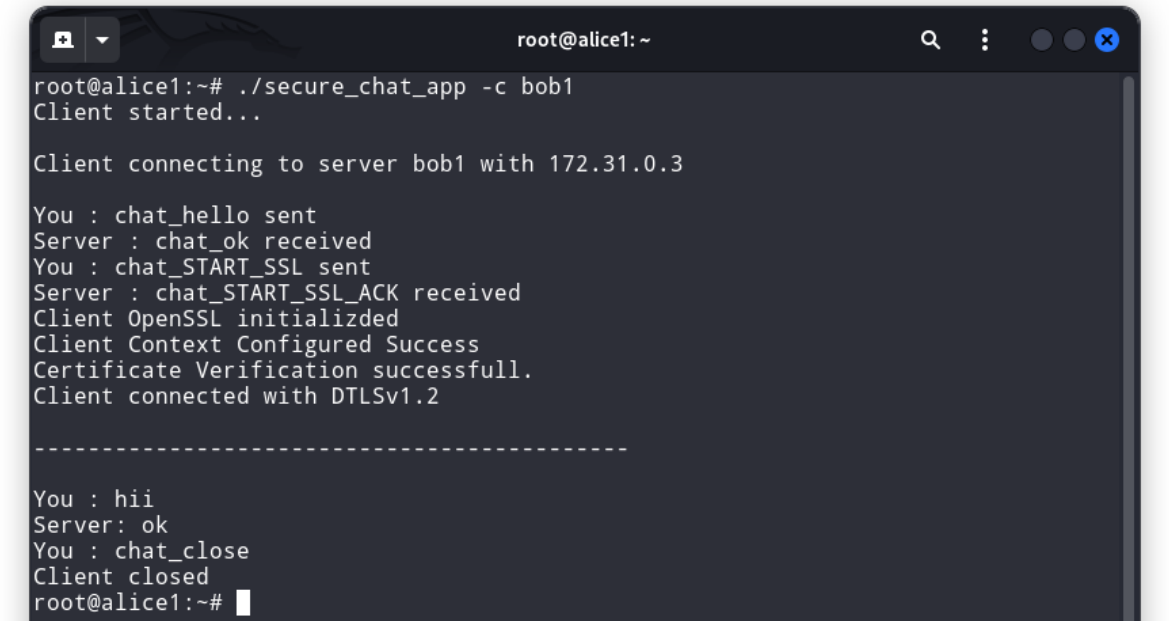
Now as part of authenticated and encrypted chatting we are verifying keys and certificate of client and server as shown in below code snippet.

```
void configure_context(SSL_CTX *ctx, const char * crt_path, const char * key_path, const char * CA_crt_path) {
        SSL_CTX_set_ecdh_auto(ctx, 1);
        if (SSL_CTX_use_certificate_file(ctx, crt_path, SSL_FILETYPE_PEM) <= 0) {
                ERR_print_errors_fp(stderr);
                exit(EXIT_FAILURE);
        }

        if (SSL_CTX_use_PrivateKey_file(ctx, key_path, SSL_FILETYPE_PEM) <= 0 ) {
                ERR_print_errors_fp(stderr);
                exit(EXIT_FAILURE);
        }

        if(!SSL_CTX_check_private_key(ctx)){
                cout << " Private Key Verification failed!\n";
                exit(0);
        }

        if (!SSL_CTX_load_verify_locations(ctx, CA_crt_path, NULL)) {
                ERR_print_errors_fp(stderr);
                exit(EXIT_FAILURE);
        }

        SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT, NULL);

        SSL_CTX_set_security_level(ctx, 1);
        SSL_CTX_set_cipher_list(ctx,"ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA384");
        SSL_CTX_set_mode(ctx,SSL_MODE_AUTO_RETRY);
        SSL_CTX_set_session_cache_mode(ctx,SSL_SESS_CACHE_OFF);
        SSL_CTX_set_verify(ctx,SSL_VERIFY_PEER,NULL);
        SSL_CTX_set_session_cache_mode(ctx, SSL_SESS_CACHE_SERVER);
        SSL_CTX_set_session_id_context(ctx, (const unsigned char *)"DTLS", strlen("DTLS"));

        if(mode=='s'){
                SSL_CTX_set_cookie_generate_cb(ctx,generateCookie);
                SSL_CTX_set_cookie_verify_cb(ctx,&verifyCookie);
        }
}
```

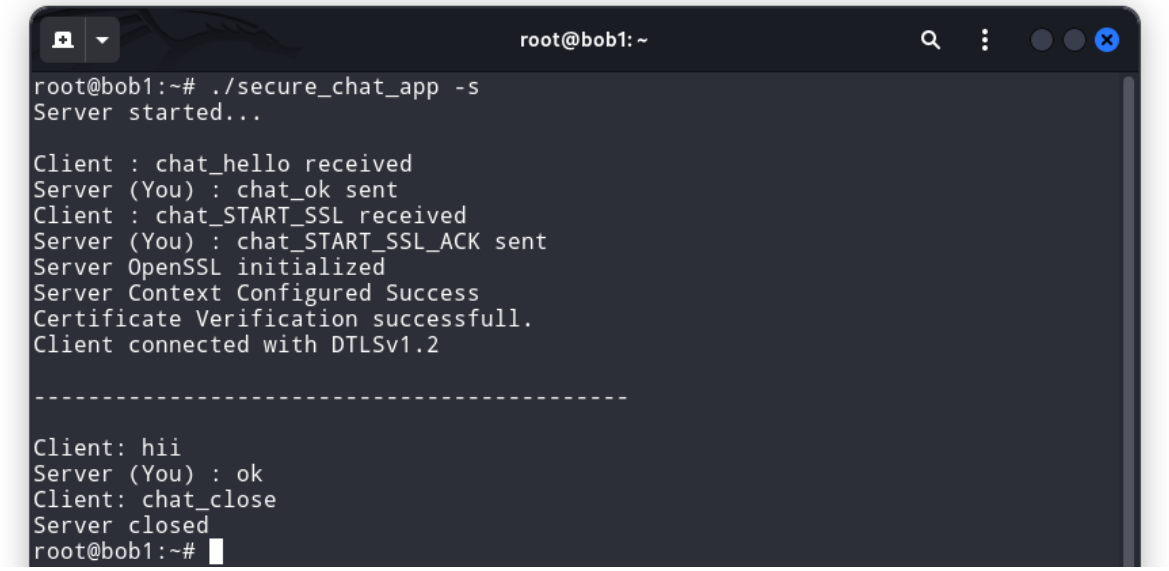
Now as a part of reliability after setting socket in not blocking mode we are using select function to listen if any activity is their till 5 sec or timeout it will wait otherwise it print Timeout and start timer again and wait for incoming activity. If any message arrives then it exit from condition and go to printing and rest part.

```
    fd_set readfds;
    FD_ZERO(&readfds);
    FD_SET(socket_descriptor, &readfds);
    timeout.tv_sec = 5;
    timeout.tv_usec = 0;
    int activity = select(socket_descriptor + 1, &readfds, NULL, NULL, &timeout);
    if (activity < 0) {
            perror("Error in select\n");
            exit(EXIT_FAILURE);
    } else if (activity == 0) {
            printf("Timeout occurred. No data received.\n");
            continue;
    }
    int received_message_len = SSL_read(ssl, buffer, sizeof(buffer) - 1);
```

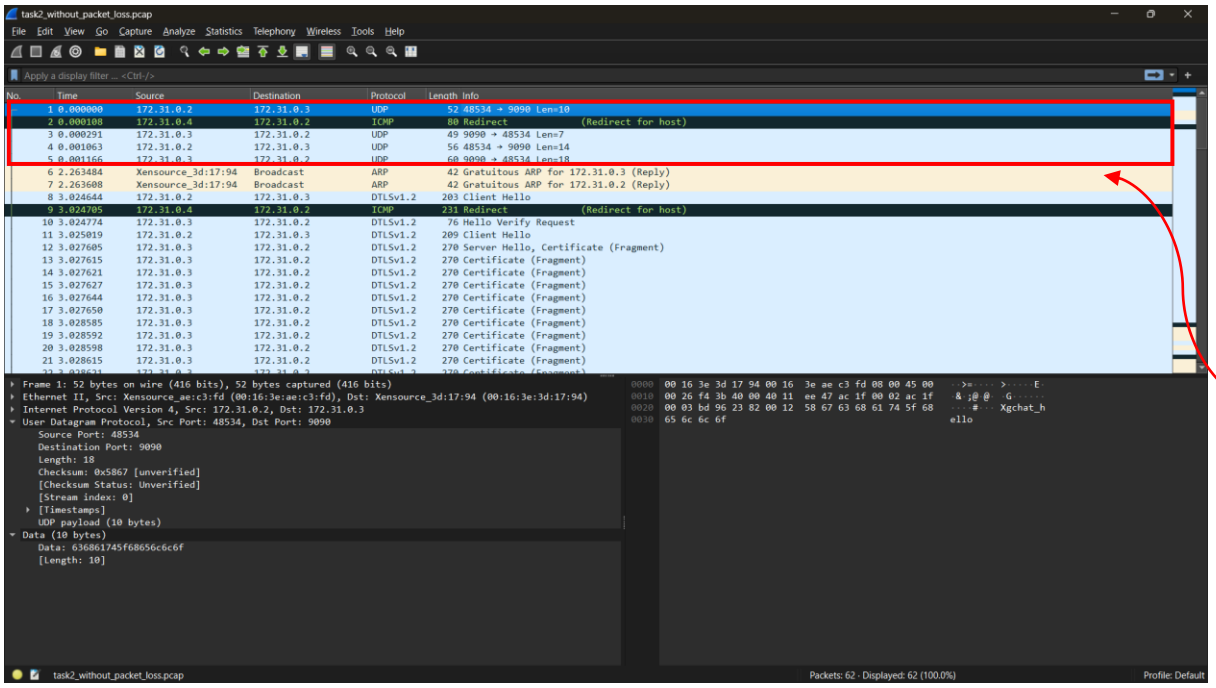
Client-Side actual output of secure chat application.



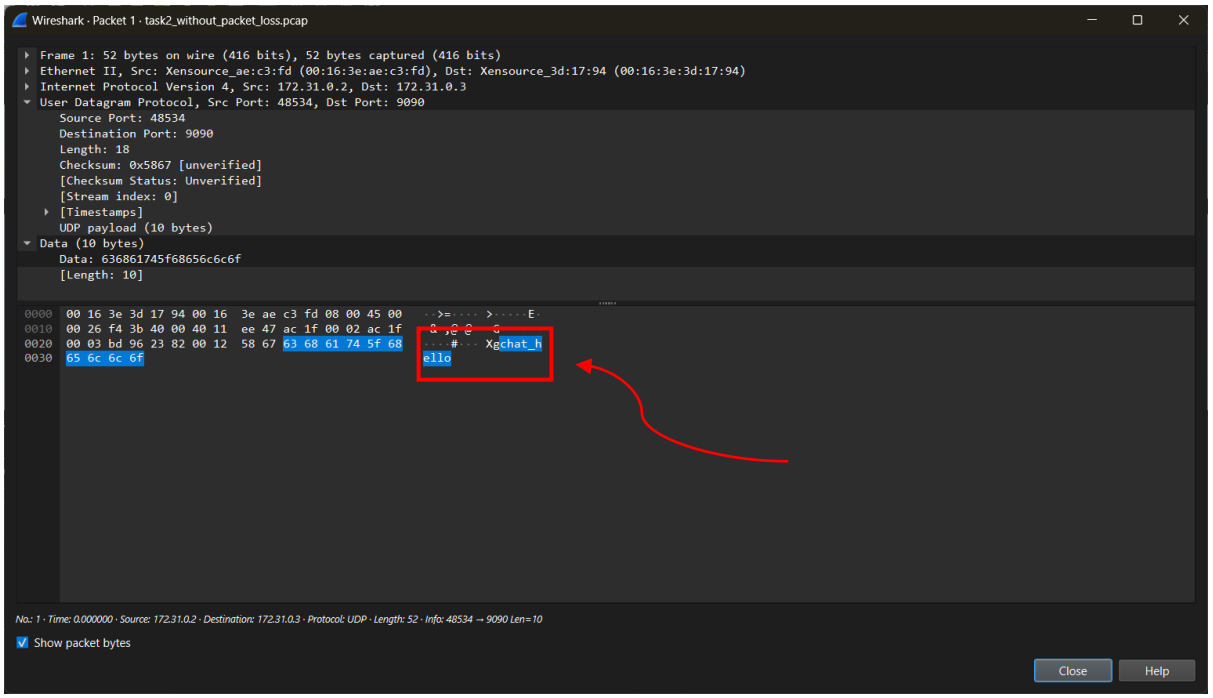
Server-side actual output of secure chat application.



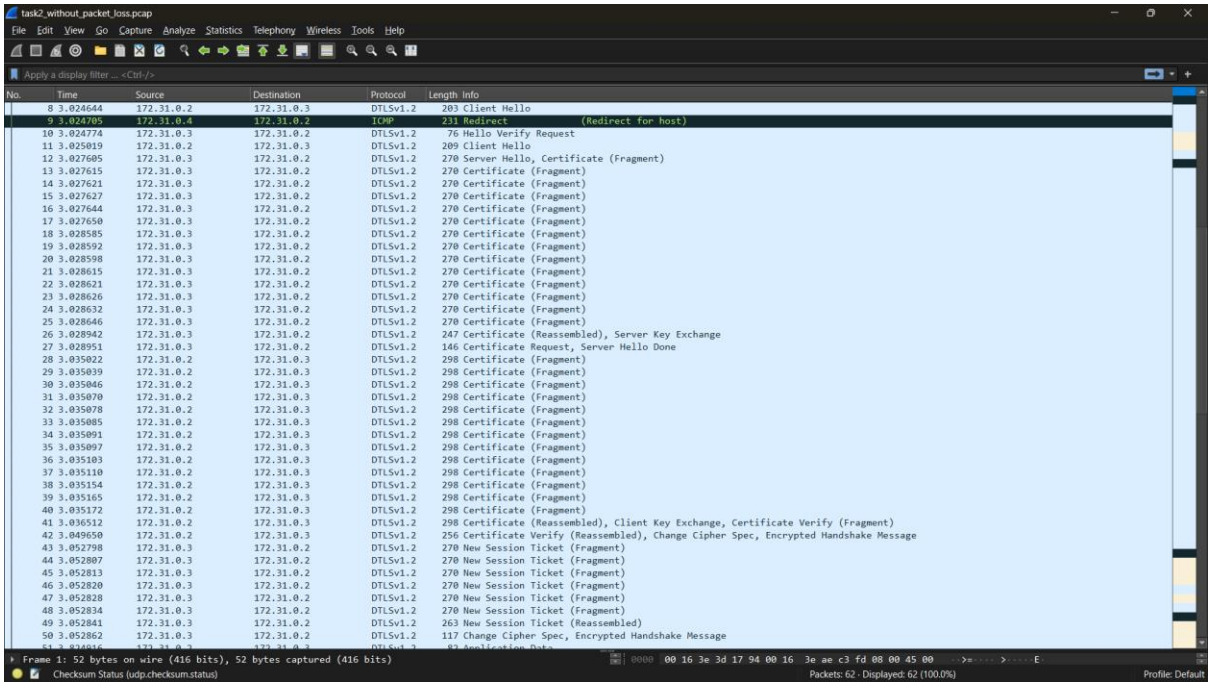
As our control message are goes under UDP socket, we can see the message by capturing it in packet capturing tool. Here below image shows the captured packets.



Below image shows actual control string message.

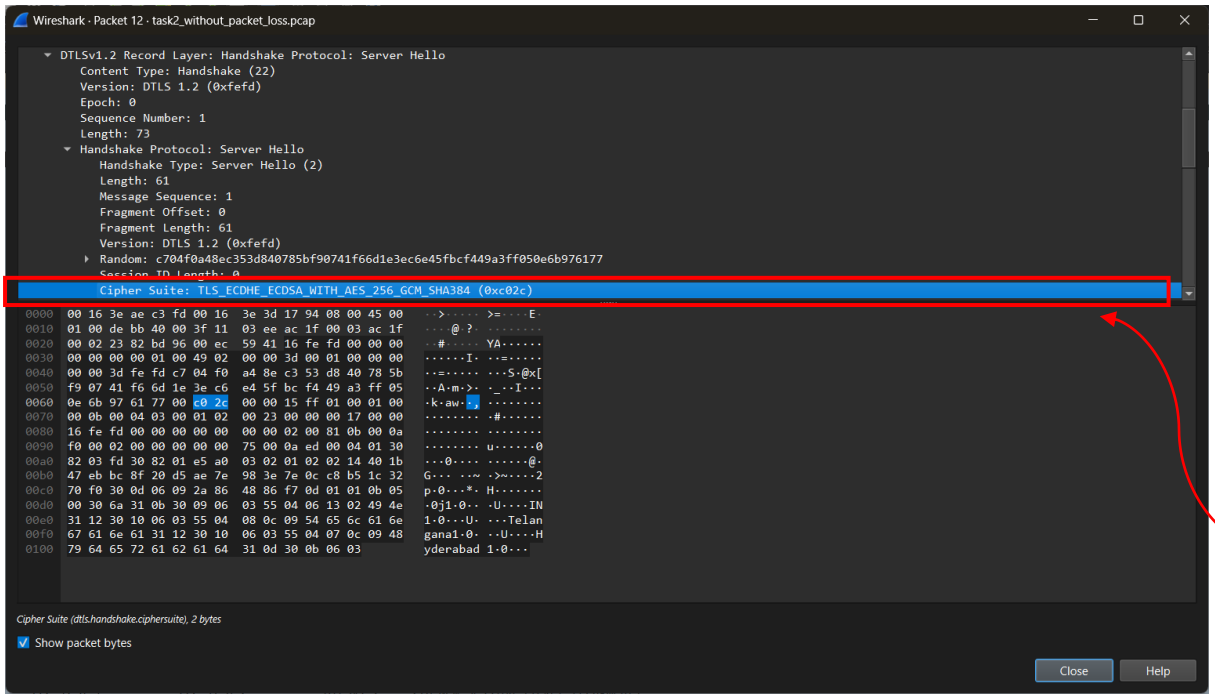


Below image shows the packet capture of DTLS v1.2 handshake.

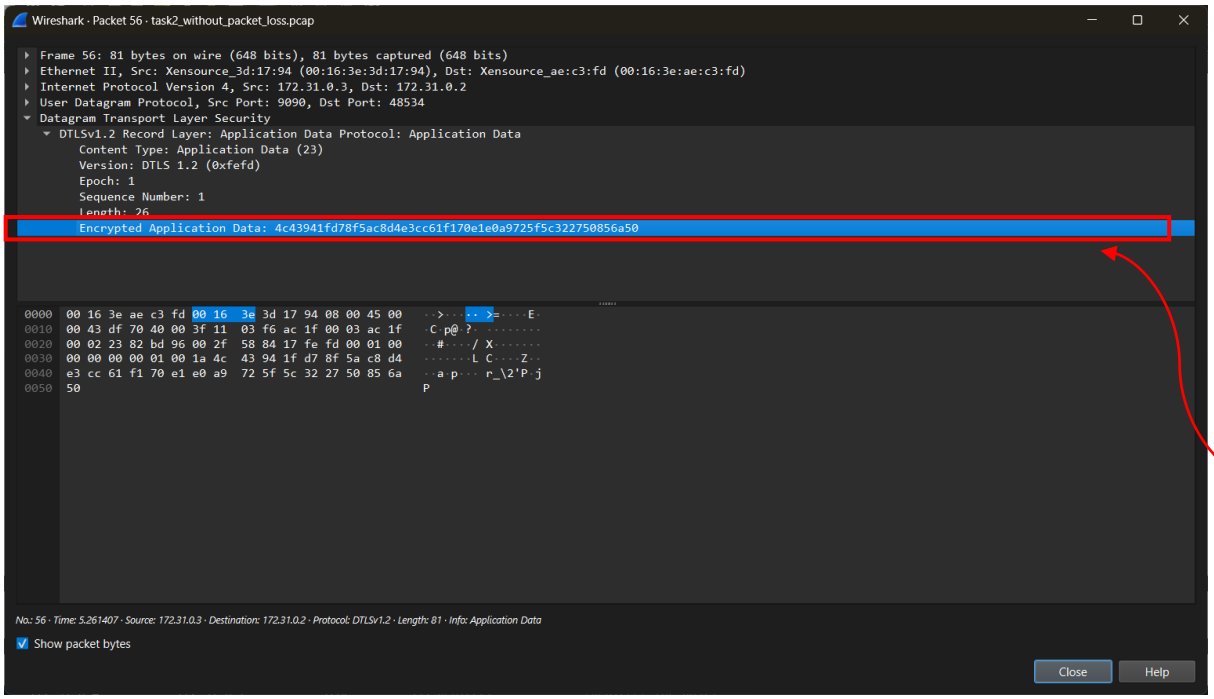


As a part of **Session Resumption** our server is sending New Session Ticket to client.

Here in below image we can see client and server agrees on **Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384** which is a one of the best cipher suite which provides **Perfect Forward Secrecy**.



Below given image shows the Packet capture of Application data which is shared after establishment of DTLS v1.2 pipe. We can see the data is in encrypted form.



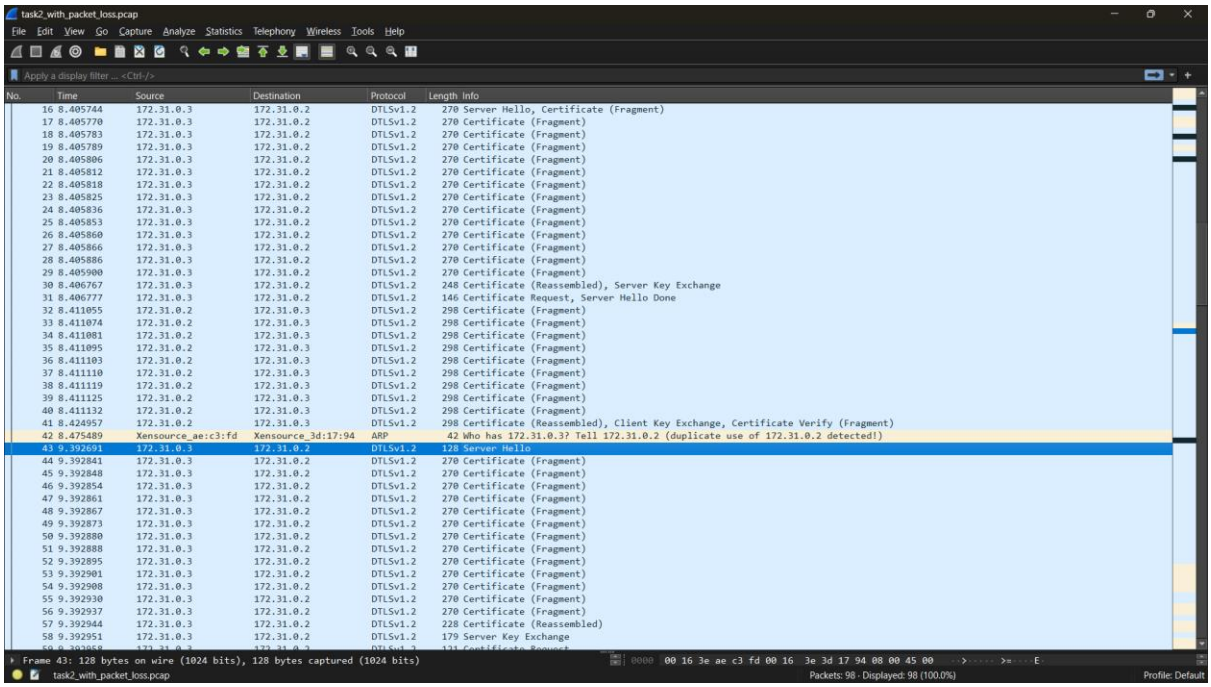
Comparing DTLS 1.2 & TLS 1.2

	DTLS v1.2	TLS v1.2
Base Protocol	It works over UDP.	It works over TCP.
Delay	It is slow compared to TLS as it works over UDP it has to manage loss and retransmission.	It is fast compared to DTLS as underlaying layer TCP handles loss and congestion control.
Handshake message dependency	Client and server can share handshake message independently which helps in out-of-order delivery of packets.	Client and Server handshake message are dependent and must be shared in a serial form.
Usage	Used for real-time applications like streaming.	Used where the reliability is a necessity.

For **testing reliability** of our application, add packet loss using netem tool with Traffic control (tc).

Command : sudo tc qdisc add dev [interface] root netem loss [loss in percentage]

We can see in the below image that the Packets from Server – Client was dropped and our chat app is able to retransmit the packets after timeout because of which Server Hello, and certificate sharing is done twice here.

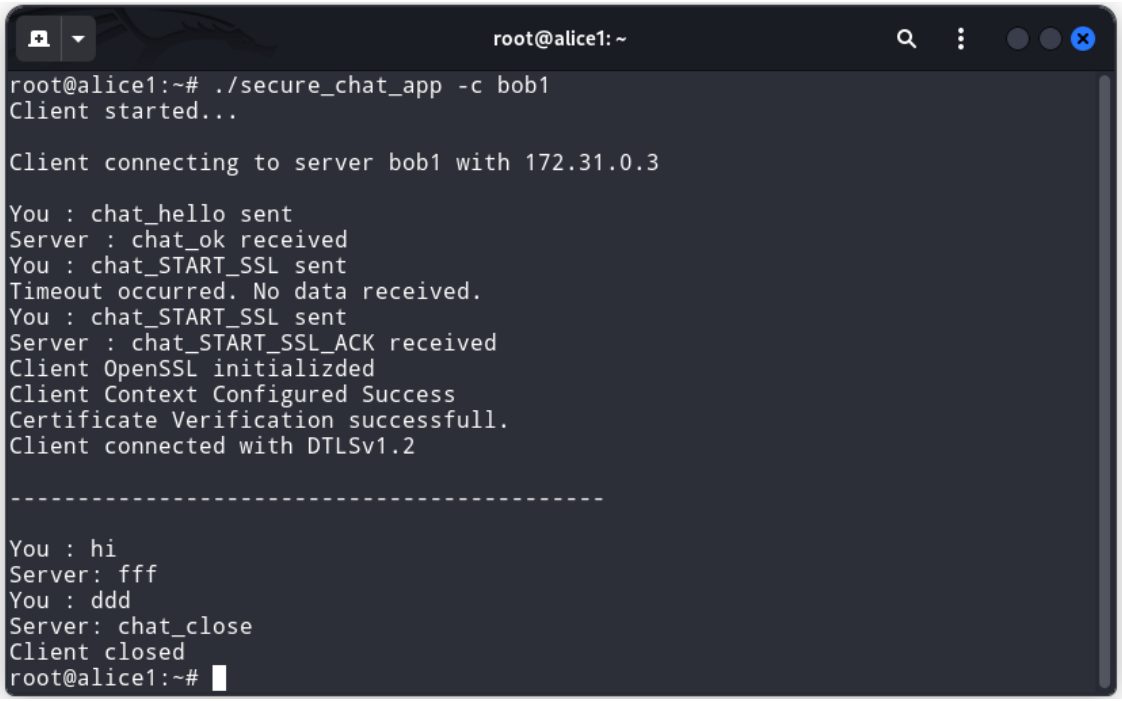


How reliability works here in our application :

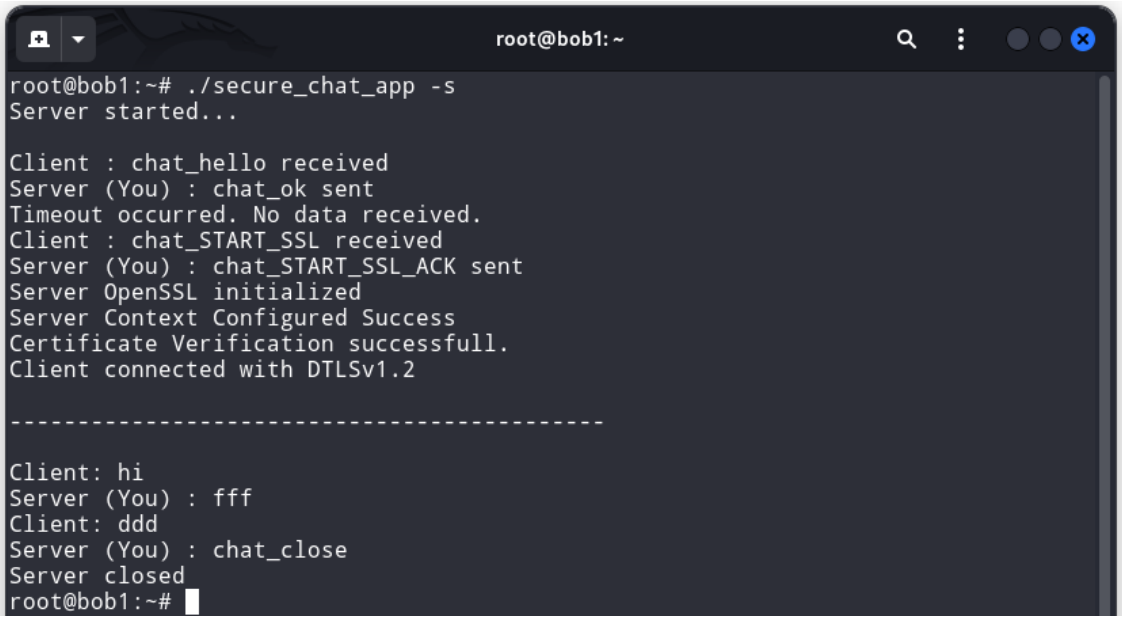
In our case in simple UDP communication we have added the timer while sending message from both server and client side. Now when any one sends the message timer starts with timeout = 5 sec, and if no ack or msg received within that time period, that particular message is retransmitted.

In DTLS communication we setup Nonblocking mode in our socket so that if message is coming then it receives else it will re-enter the loop and try to receive message for timeout period and if message not received then it prints Timeout and re-enter loop and received again for timeout period.

Client-Side actual output of secure chat application with 20% packet loss.



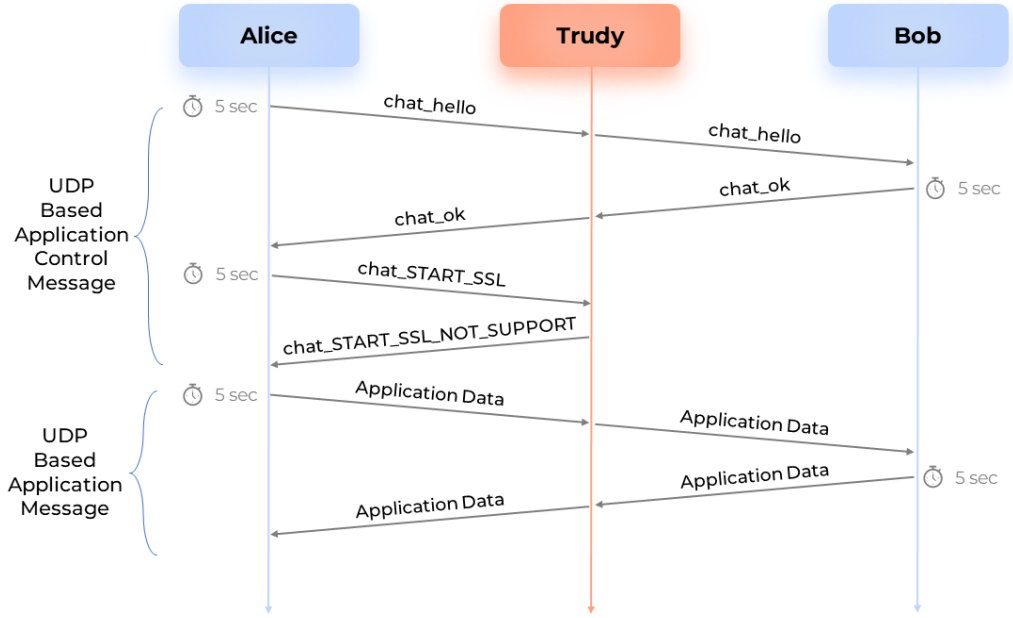
Server-Side actual output of secure chat application with 20% packet loss.



Task 3 – SSL Downgrade Attack for eavesdropping

Assuming Trudy has successfully poisoned /etc/hosts file.

Application Flow :



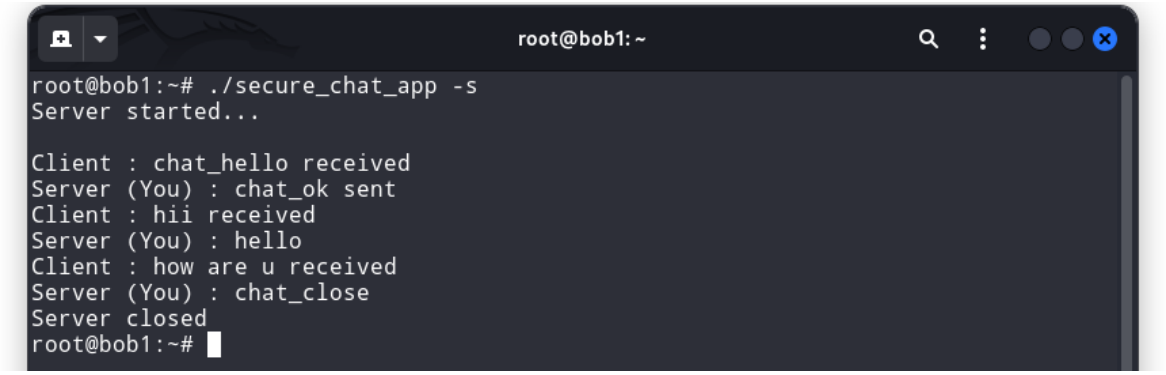
Above exceptional flow diagram created by one of our team member shows the basic working of our chat application with SSL downgrade attack.

Here Trudy intercept all messages from Alice to Bob, Now when Alice sends the chat_START_SSL, Trudy block that message and reply it as chat_START_SSL_NOT_SUPPORTED which Alice understand that it is coming from Bob and Bob doesn't able to support SSL communication so Alice continues with the UDP open communication.

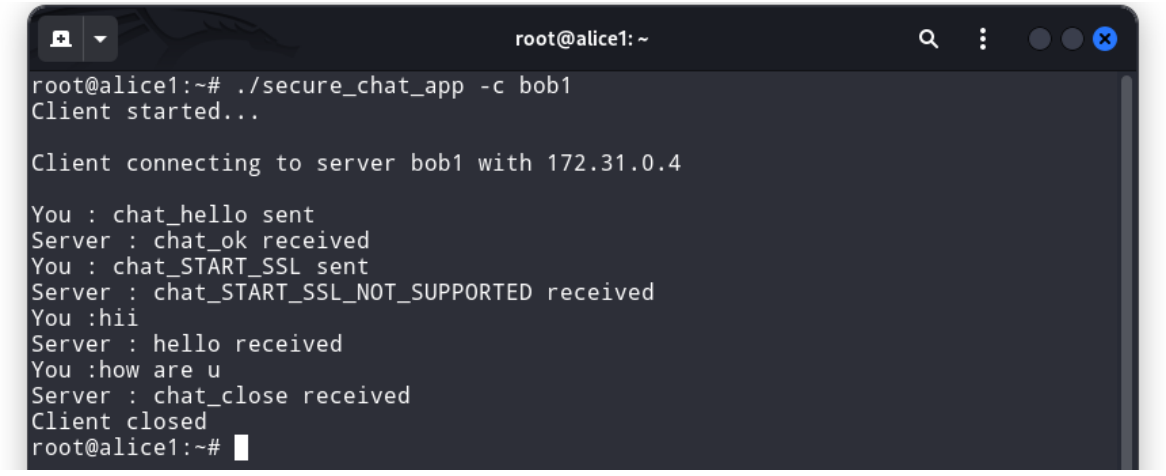
This code snippet given below consist of main logic behind the downgrading at Trudy's side where it checks every received message and if chat_START_SSL is received it sends chat_START_SSL_NOT_SUPPORTED.

```
received_message_len = recvfrom(fake_server_socket_descriptor, (char *)buffer, MAX_BUFFER_SIZE-1, 0, (struct sockaddr *) &client_address, (socklen_t *)&le
buffer[received_message_len] = '\0';
cout << "Client : " << buffer << " received from client.\n";
if(strcmp(buffer,"chat_START_SSL")==0){
        ack = "chat_START_SSL_NOT_SUPPORTED";
        goto jump;
}
msg = buffer;
sendto(fake_client_socket_descriptor, msg.c_str(), msg.length(), 0, (const struct sockaddr *) &server_address, sizeof(server_address));
cout << "Forwarding message : " << msg << " to actual server.\n";
if(msg=="chat_close"){
        break;
}

memset(buffer, '\0', sizeof(buffer));
received_message_len = recvfrom(fake_client_socket_descriptor, (char *)buffer, MAX_BUFFER_SIZE-1, 0, (struct sockaddr *) &server_address, (socklen_t *)&le
buffer[received_message_len] = '\0';
cout << "Server : " << buffer << " received from server.\n";

ack = buffer;
jump:
sendto(fake_server_socket_descriptor, ack.c_str(), ack.length(), 0, (const struct sockaddr *) &client_address, sizeof(client_address));
cout << "Forwarding message : " << ack << " to actual client.\n";
if(ack=="chat_close"){
        break;
}
```

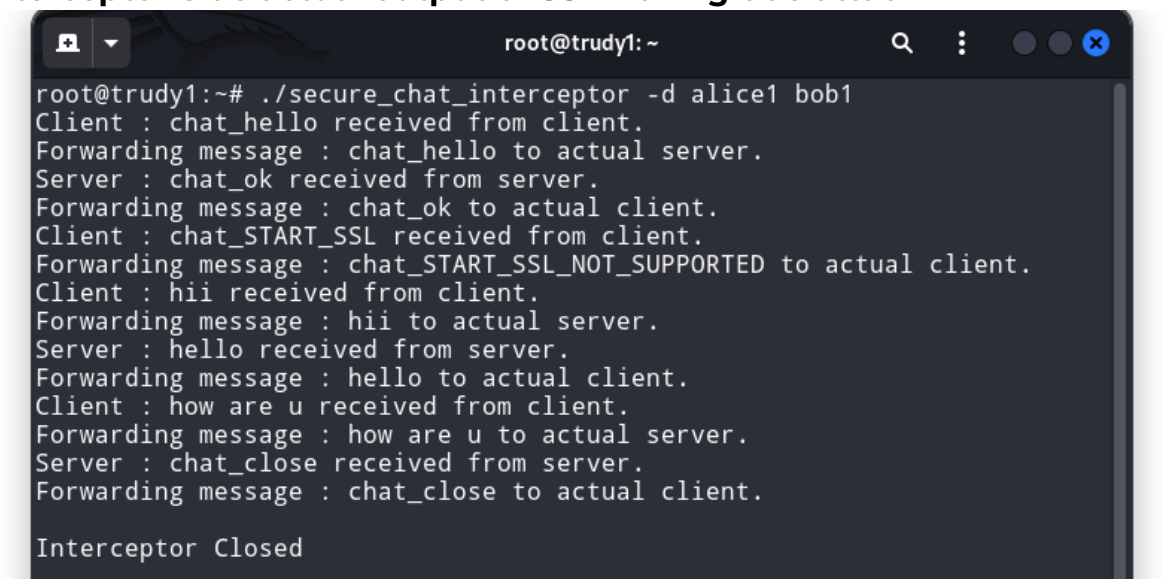
Server-side actual output of secure chat application.



Client-side actual output of secure chat application.



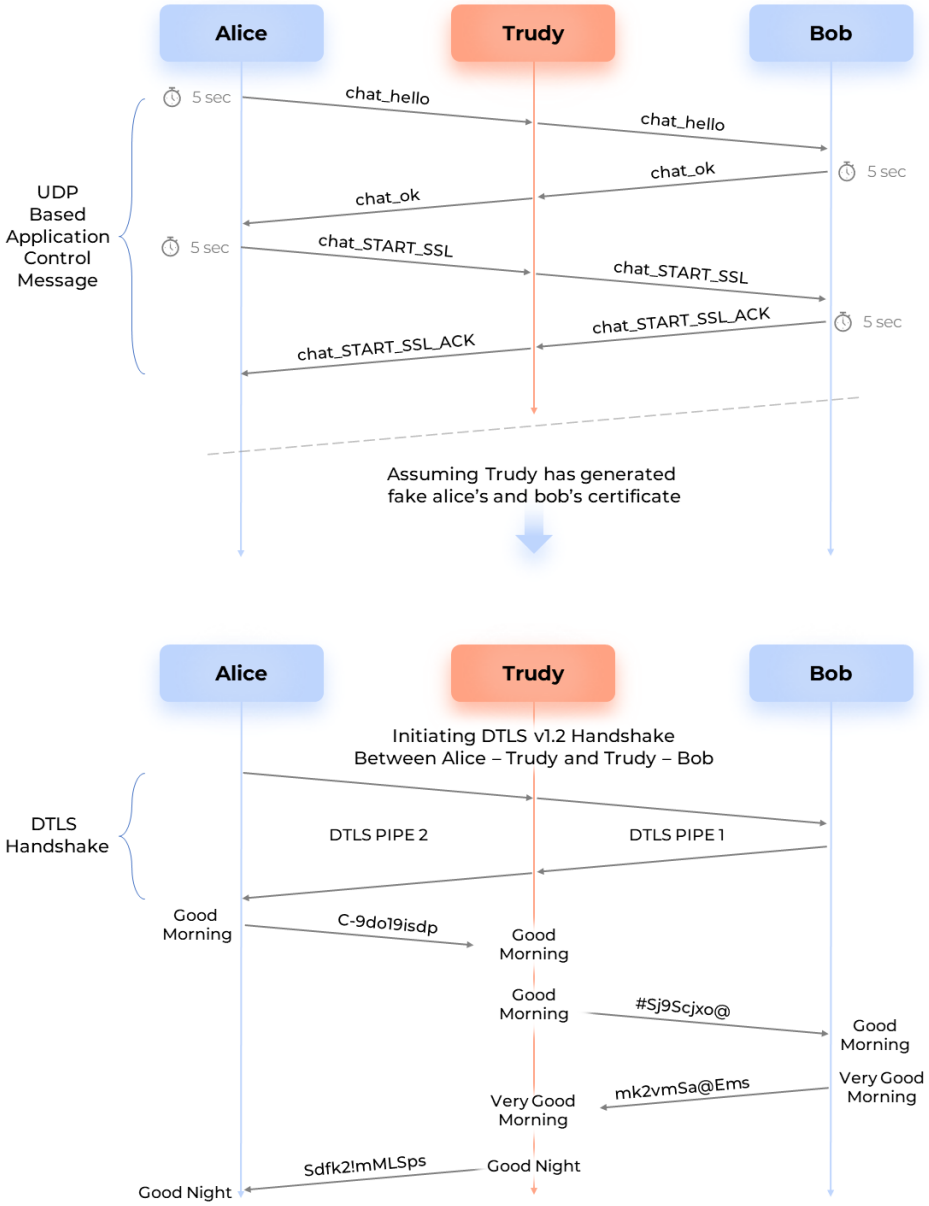
Interceptor-side actual output of SSL Downgrade attack



Task 4 – Active MITM Attack

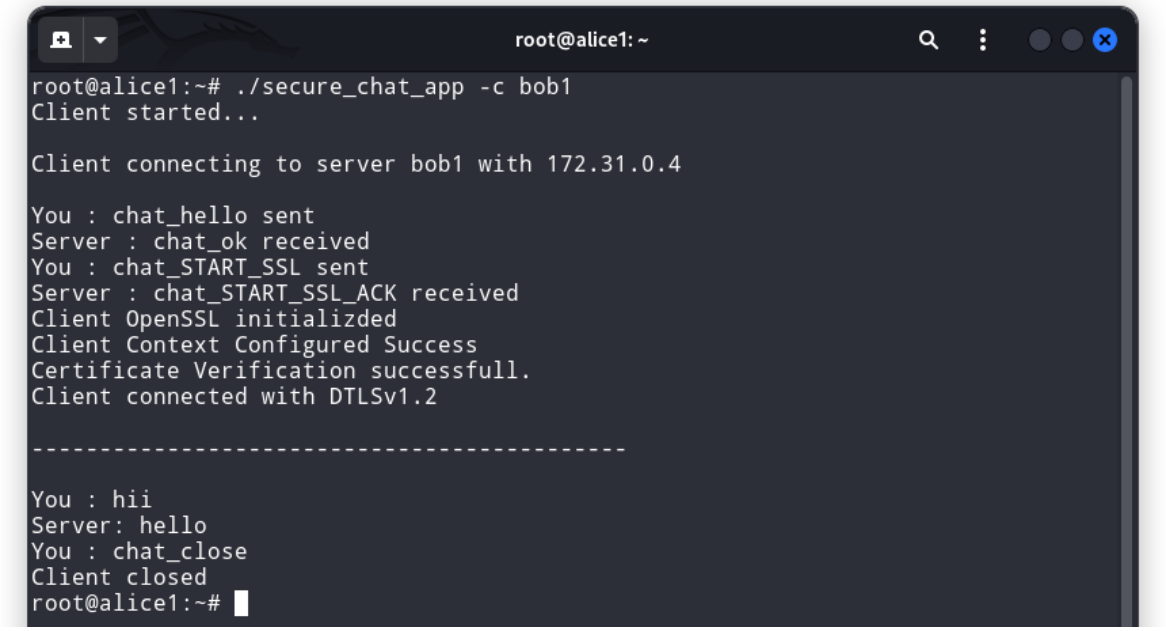
Assuming Trudy has successfully poisoned /etc/hosts file.

Application Flow :

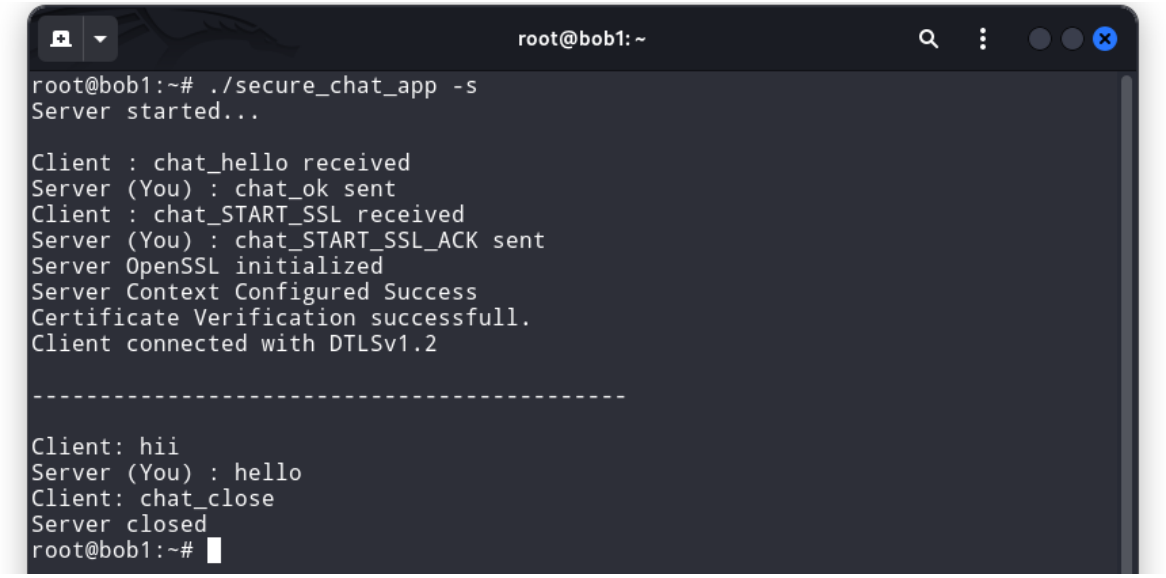


Above exceptional flow diagram created by one of our team member shows the basic working of how Active MITM attack is performed here.

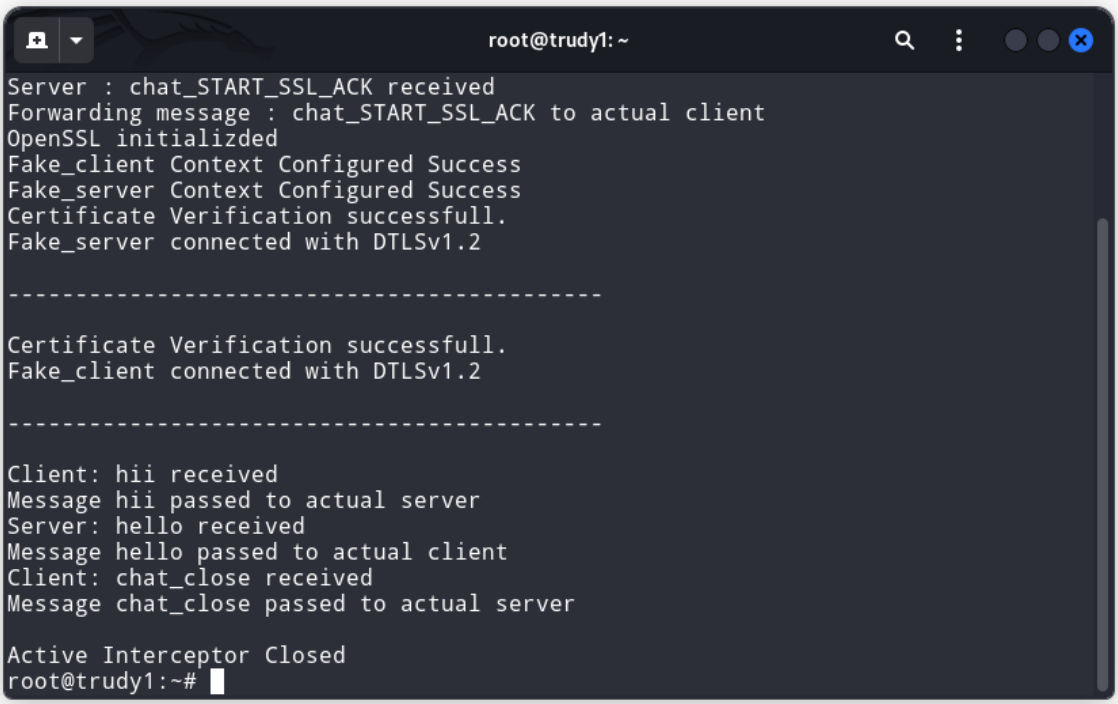
Alice-side actual output of Active MITM attack



Bob-side actual output of Active MITM attack



Trudy-side actual output of Active MITM attack



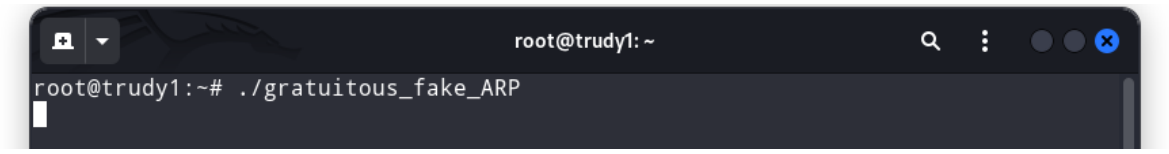
We are implementing MITM in two mode if Interceptor has to modify message then he/she enter the modified string or if he/she don't want to modify then simply enter will send the original content as shown in below snippet.

```
}
received_message_len = SSL_read(ssl_server, buffer, sizeof(buffer) - 1);
if (received_message_len <= 0) {
        int ssl_error = SSL_get_error(ssl_server, received_message_len);
        if (ssl_error == SSL_ERROR_ZERO_RETURN || ssl_error == SSL_ERROR_SYSCALL) {
                break;
        }
} else {
        buffer[received_message_len] = '\0';
        cout << "Client: " << buffer << " received\n";
}
string modified;
cout << "Type to modify else Enter : ";
getline(cin,modified);
if (modified == ""){
        msg = modified;
}
else{
        msg = buffer;
}
SSL_write(ssl_client, msg.c_str(), msg.length());
cout << "Message " << msg << " passed to actual server\n";
```

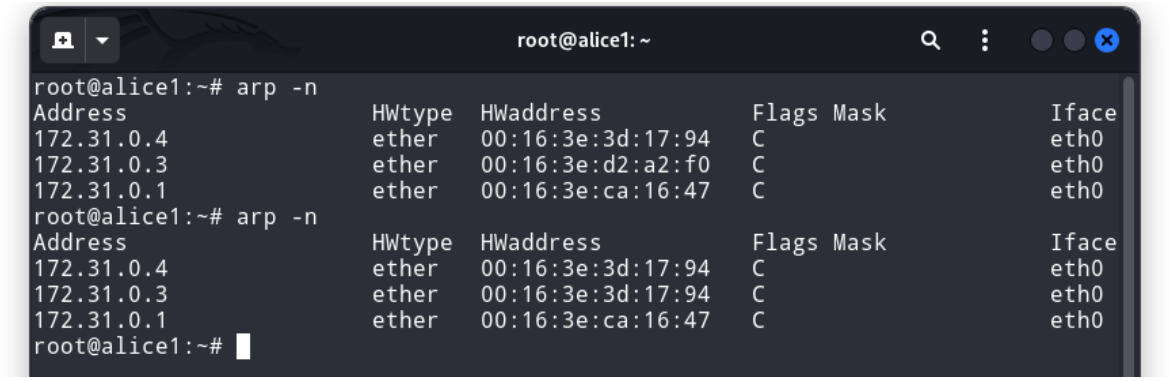
Task 5 – ARP Cache Poisoning

For ARP cache poisoning we are creating a fake ARP response packet and broadcast it using gratuitous_fake_ARP.cpp

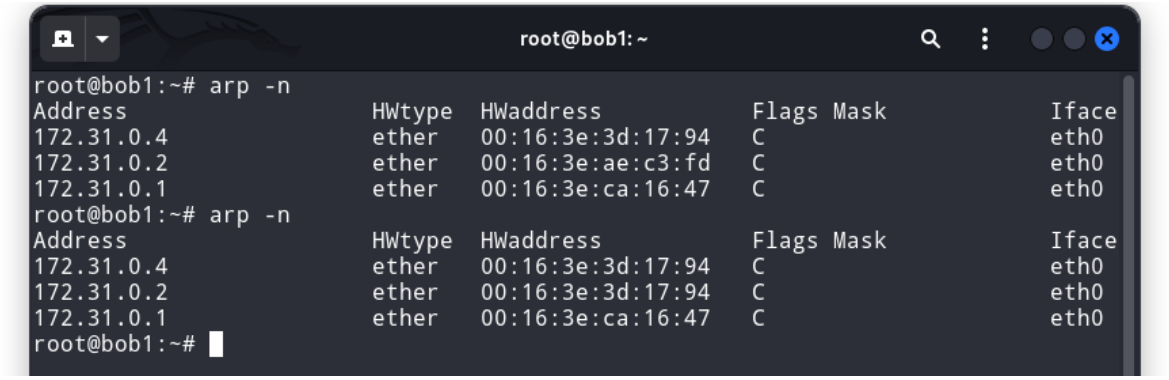
Sending Fake Gratuitous ARP reponse packet from Trudy



We can see before and after sending the fake ARP that at alice's side MAC address corresponding to 172.31.0.3 got changed with Trudy's MAC.



Also at Bob's side, we can see before and after sending the fake ARP that MAC address corresponding to 172.31.0.2 got changed with Trudy's MAC.



Major Libraries Used

- Openssl : For DTLS
- pcap.h : For generating ARP response packet (Task 5)

Tools Used

- Tcpdump : For packet capturing
- Microsoft Powerpoint : For creating flow diagram
- Microsoft Word : For report
- After Effect : For video editing

ANTI-PLAGIARISM STATEMENT

We certify that this assignment/report is our own work, based on our personal study and/or research and that we have acknowledged all material and sources used in its preparation, whether they be books, articles, packages, datasets, reports, lecture notes, and any other kind of document, electronic or personal communication. We also certify that this assignment/report has not previously been submitted for assessment/project in any other course lab, except where specific permission has been granted from all course instructors involved, or at any other time in this course, and that we have not copied in part or whole or otherwise plagiarized the work of other students and/or persons. We pledge to uphold the principles of honesty and responsibility at CSE@IITH. In addition, we understand my responsibility to report honor violations by other students if we become aware of it.

Names: Patel Heetkumar D, KR Anuraj, Vishal Patidar

Date: 10/04/2024

Signature: HD, Anuraj, VP