LAB #3: SECURE CLIENT-SERVER
DUE DATE: Friday April 22, 11:59 p.m.

TO BE DONE IN GROUPS OF THREE

1. General Description

In this laboratory, you will implement a secure client-server system. You may leverage your code from Lab1 and Lab2. There are two key pieces to this laboratory: (i) authentication of the client and server using a simplified Kerberos protocol; and (ii) file transfer from server to client, with transmission encrypted by the server.

2. File with header files

We will provide you with a header file packet.h that contains the format information of all the packets, and defines the types of all the fields. You should include this header file in all your C programs, and carefully read the information present in the comments. There is also a README file that contains other important implementation details. These two files define all details you need to complete the assignment, and the instructions will guide your design. Please download the file lab3.tar.gz from: http://shay.ecn.purdue.edu/~ece495f/labs/lab3/lab3.tar.gz

3. Server – Client application:

Requirements:

There are three pieces of code you have to implement in three different files -- a client, an authentication server and a server. In this handout, we will give you the high level details of the protocol. For detailed packet type information, please refer the header.h file and README file provided as part of the assignment.

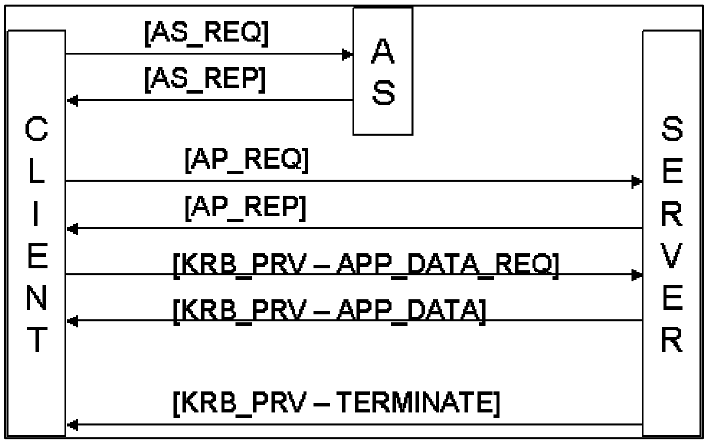

Figure #1 Functional specification

Authentication:

In the description that follows, Kc, and Ks are secret keys shared by the Authentication Server with the client and the server respectively. Kc,s is a secret key generated by the Authentication Server, and shared between the client and the server. Furthermore, the packet types that contain the information for each step of the authentication protocol will be presented with the description below (Refer to Figure #1):

a. The client sends a message to the AS of the form (AS_REQ message):
   <ClientID> <ServerID><TimeStamp1>

b. The AS responds to the client with a message of the form (AS_REP message):
   E_{Kc}[Kc,s || ServerID || TimeStamp2 || Lifetime2 || Tkt]

c. The Tkt is of the form:
   E_{Ks}[Kc,s || ClientID || ClientIP || ServerID || TimeStamp2 || Lifetime2]

d. The client sends the server a message of the form (AP_REQ message):
   Tkt || Authenticator

e. The Authenticator is of the form:
   $E_{Kc,s}$[ClientID || ClientIP || TimeStamp3]

f. The server returns a message to the client of the form (AP_REP message):
   $E_{Kc,s}$[TimeStamp3+1]

Data Transmission:
The server transmits data to the client, encrypted with the secret key Kc,s that was established between them during the authentication process. The packet types that contain the information for each step of the data transmission protocol will be presented with the description below (Refer to Figure #1). Please note that each of the packets mentioned below is encrypted and stored in a KRB_PRV packet type and then transmitted through the network:

a. The client transmits an APP_DATA_REQUEST packet to the server.
b. Once the server verifies the packet type to be a APP_DATA_REQUEST, it reads the file to be transmitted. The file is broken into several segments (depending on the size of the file), and each segment is stored in a APP_DATA packet. This packet is then encrypted using the secret key Kc,s, using a DES-CBC encoding, and using an IV of 0 (the IV is a 8-byte character array, all bytes of which are set to 0).
c. When the server completes sending the file, it will transmit a TERMINATE packet which marks the end of the file download. Included in this packet, is a file digest (SHA1 digest) that the client will use to verify the integrity of the received file.

Error packets
Two extra packets should be included as part of your implementation:

a. AS_ERR: Will be sent from the Authentication Server to the client in case the Client ID and Server ID provided by the client in the AS_REQ message do not match the Client ID and Server ID stored at the Authentication Server. In case the client receives this message, it should gracefully exit the protocol. The Authentication Server should also exit the protocol.
b. AP_ERR: Will be sent by either the client (to the AP) or the AP (to the client) in two situations:
    1. If the client received a wrong AP_REP message from the AP, it will return a AP_ERR to the AP. Recall that the AP_REP message contains the value (timestamp3 + 1), where timestamp3 was initially sent from the client to the AP.
    2. If the AP receives a client id or client ip address from the Authenticator that do not match with the client id and client ip address that were sent with the ticket by the AS

Notes:

a. You need to include the header file that we have provided in all your C-files. You will also be able to access information regarding all packet types, and formats of fields there.

b. Like in Lab2, you may assume that the server handles only one client at a time and need not address issues associated with supporting multiple simultaneous clients. Further, there is no need for you to handle error recovery with UDP, however, in the event that a packet loss does occur, the code must exit gracefully (for example the code should detect a gap in data sequence number, or receive an unexpected packet)

c. For all encryption, use DES-CBC. We recommend that you use the built-in function ***DES_ncbc_encrypt*** , information regarding which is present in the README. For all encryption/decryption with DES-CBC, you may assume an Initialization Vector (IV) of 0.

Command Line Arguments:

Your authentication server code must be executed from the command line as follows:

./authserver <authserverport> <clientID> <clientkey> <serverID> <serverkey>

<clientID> and <serverID> are strings not to exceed a length of 40 characters. <clientkey> is the secret key shared between the authentication server and the client, and <serverkey> is the secret key shared between the authentication server and the server. The keys must be represented as a string comprised only of hexadecimal digits. Please refer the README for format of <clientkey> input.

The server code must be executed from the command line as follows:

./server <server port> <authserverkey> <input file>

<authserverkey> is the secret key shared between the server and authentication server. The keys must be represented as a string comprised only of hexadecimal digits. Please refer the README for format of <authserverkey> input. The input file is the path to the file that the server will send to the client.

The client code must be executed from the command line as follows:

./client <authservername> <authserverport> <authserverkey> <server name> <server port> <output file> <clientID> <serverID>

The <authserverkey> is the secret key shared between the client and the authentication server. The keys must be represented as a string comprised only of hexadecimal digits. Please refer the README for format of <authserverkey> input.
The output file argument is the file name to assign to the file the server will send to the client. <clientID> and <serverID> are strings not to exceed a length of 40 characters.

If any of the arguments is invalid, your code should return an appropriate message to the user. Be sure to consider the case when the keys are invalid.

Output messages:

You must print to the screen (STDOUT) the following messages in case your application finishes correctly or terminate unexpectedly:

a. Print the message "OK" in case your application finishes correctly. This is the case when the server is able to completely send the file to the client and the digest sent by the server matches the digest of the file received by the client.
b. Print the message "ABORT" in case an error occurs, followed by a description of why the error occurs, Example erroneous situations are: (i) the digest of the file the client receives differs from the digest sent by the server (ii) the client or server receives an unexpected packet

There could be other situations where it becomes necessary to abort the protocol. You should be able to detect this situation and print out the correct message.

4. Deliverables

a. A physical copy of the files client.c, server.c, and authserver.c submitted using the "turnin" command. Name your files <login1>.<login2>.client.c <login1>.<login2>.server.c, and <login1>.<login2>.authserver.c where login1 and login2 are login names of the two members of the group. For example: sanjay.rtorresg.client.c
    - *Submission instructions*
        - *Log into shay.ecn.purdue.edu*

  - *Move to the same directory as the file you are going to submit.*
  - *Execute: "turnin –c ece495f –p lab3 <login1>.<login2>.client.c"*
  - *Execute: "turnin –c ece495f –p lab3 <login1>.<login2>.server.c"*
  - *Execute: "turnin –c ece495f –p lab3 <login1>.<login2>.authserver.c"*

b. Printout.

Turn in a **typed** report that includes the C-code of the 3 files above.

5. Grading:

Grading will involve a live demo with each group. We will try to provide binary versions of our implementations so you can ensure your implementation conforms to ours. More details will be sent by e-mail.

6. Milestones/Checkpoint:

A checkpoint of your code is due by Friday, Apr 15th, 11:59pm.