ΕΠΛ 674 Ασφάλεια Συστημάτων και Δικτύων	**Spring 2011**
	Assignment 4
Assigned: 14 March 2011	**Due: 4 April 2011**

Instructions

- All assignments should be submitted typed neatly using a document processing application of your choice. Please make sure to include your name and student number for proper recording of grades.
- Assignment solutions can be written English.
- The assignment is due at the beginning of the lecture at the due date. Late assignments will incur a five-point penalty. Assignments late by more than one day will not be accepted.

PART A – Client-Server System

This is a TCP sockets programming exercise in C. If you can't do it, you may not have adequate background for the programming assignments to follow in this class.

In this assignment you will develop a simple CHAT server.
The primary objectives of this assignment are:

- to illustrate the client-server paradigm;
- to show you how programs actually access network services;
- to introduce you to the UNIX/C sockets interface.

1. Setup and Preparation

To complete this assignment, you will have to use a unix-based system (unix, aix, linux, solaris). This assignment requires that you have some understanding of the use of the sockets API for connection-oriented interprocess communication.
To that end, you will use the course lectures and you should also need to read carefully the manual ("man") pages for the socket(), connect(), send(), recv(), close(), bind(), listen(), accept(), and shutdown() routines, if you are not already familiar with them. (you will need the manual pages for details.)

2. The Protocol

The client and server communicate by exchanging lines of printable ASCII characters via the reliable byte-stream service provided by TCP. (For this lab, the socket type is SOCK_STREAM, and the protocol family is PF_INET.)

The interaction between the client and server for this exercise proceeds as follows:

1. The server "listens" for connections on some port (above 1024). You may use the port 15001.
2. The client opens a connection to the server's socket.

3. The server accepts the connection and waits to receive a request string from the client.
4. Once the client is connected to the server, it immediately sends a request string. The format of the request string is:

   EPL674 <WS> <request type> <WS> <username> <WS> <endpoint-specifier> <newline>

   where:

   <WS> is "whitespace", one or more blank or tab characters.

   The <request type> is a string. For this exercise use "chat".

   The <username> is the student's username (you can use your CS account ID). The username must contain only printable ASCII characters, and may be no more than 64 characters in length.

   The endpoint specifier is of the form <client IP address>-<client port number> (The endpoint specifier in this exercise refers to the client's socket.)

   <newline> is the end-of-line marker, namely the two-character sequence "\r\n".

5. Thus, if the client's socket is bound to port 12981 on 128.163.1.163, the request string looks like: EPL674 chat gpapadim 128.163.1.163-12981\r\n
6. Note that the maximum length of the request string, excluding whitespace and the terminating \r\n sequence, is 96 characters.
7. Upon receiving and parsing the request string, the server responds by sending a greeting terminated by "\r\n". If the request string was correctly formed, the line looks something like: Hello gpapadim from 128.163.1.163-12981. Welcome to the EPL674 chat room\r\n
8. If the request is not properly formed, or does not contain the proper endpoint identifier, the line will contain an appropriate error message.
9. After receiving the greeting line, **the CLIENT will then loop until end-of-file reading from both stdin and the network connection**. Anything it sees on stdin, it copies to the server; anything it sees on the network connection from the server, it writes to stdout.
10. After sending the greeting line, **the SERVER will then loop until end-of-file reading from the network connection**. Anything it sees on the network connection from the client, it writes it back to the network connection (echo operation). When the client closes the connection, the server waits for new connection.

Client Operation

To implement the above protocol, the client does the following:

(i) Create a socket (of type SOCK_STREAM and family PF_INET) using socket().

(ii) Fill in a sockaddr_in data structure with the host IP address and port number respectively) of the EPL674 server. To convert a "dotted quad-port" string to a socket address, i.e. to fill in a sockaddr_in structure, use the inet_addr() (3N) routine to convert a "dotted quad" string to the correct 32-bit unsigned integer representation, and atol() (3) to convert the port number.

(iii) Call connect() with the filled-in sockaddr_in data structure to initiate a connection to the server. If unsuccessful, print out the reason for the error and exit.

(iv) Construct a request string using the client's address information. To determine the client address information, you will need to use the getsockname() routine (see the man page). You may want to use sprintf() to format the request line.

(v) Send the request to the server (using send()).

(vi) Read data from the socket (using recv()) until you encounter the \r\n sequence. Verify that the server accepted your request string. Then read and send/display bytes until you encounter the \r\n sequence again.

Note that what you send should look exactly as specified above. In particular, the first character of any line must be non-whitespace, or the server will complain of a syntax error. Also, you must take care not to send extra characters after the end-of-line terminator. Remember that the TCP byte-stream service doesn't know anything about null-terminated strings. That is, if you are going to use printf() to display what you receive over the network, you must make sure you put in a null character to terminate the string after calling recv().

Also note that when the socket is first created, it is not bound to any address. At connect time, however, the system implicitly assigns an address to the socket. The IP address will be the host's IP address, while the port number is chosen more-or-less arbitrarily. The point is the desired IP address information can't be obtained using getsockname() until after the socket is connected to the server.

Things to worry about include how long a line is, how to read a line without blocking, and other forms of client misbehavior.

3. **Requirements**
   You will turn in a writeup comprising your well-commented code and its output and a ZIP file containing the actual code. **Only two files are required; one for the server and one for the client**. You will be graded on (i) the correctness of your code; and (ii) its readability and structure. Note that the correctness is easier to determine if the code is readable and well commented.

PART B – Chryptographic Algorithms

The notion of cryptography consists of hiding secret information from non trusted peers by mangling messages into unintelligible text that only trusted peers can rearrange.
In this section, we will use and modify three different techniques commonly employed to hide or encrypt information: secret key cryptography (DES), public key cryptography (RSA) and message digests (SHA-1).

1. **General Description**
   You will implement and test three important cryptographic algorithms: DES, RSA, and SHA-1. There is a group of built in functions that will help you complete this assignment. These functions are included in Openssl. Openssl is an open source toolkit that implements security protocols such as SSL but also includes a general purpose cryptography library which you will use. Openssl is usually part of most of the recent linux distributions.

2. **File with sample codes and results**

You are given a group of files as a reference on how to use certain built in functions that you must use in your implementations. These files are just skeletons that you should modify in order to allow for reading from arbitrary files. Included in the compressed file given is also a test case for a DES-CBC implementation (test.txt and test.des) and a file with general descriptions of some built in functions. **Please download the file lab4B.tar for all PARTB sample files.**

3. **Requirements**
   a. Just use the built in functions that appear in the skeleton files. **Only three files are required; one for the each algorithm (DES, RSA, SHA1)**.
   b. Your code should result in an executable of the following form (example of DES):
      ./tempdes iv key inputfile outputfile

      The parameters description is as follows:
      - iv: the actual IV to use: this must be represented as a string comprised only of hexadecimal digits.
      - key: the actual key to use: this must be represented as a string comprised only of hexadecimal digits.
      - inputfile: input file name
      - outputfile: output file name

      For example:
      ./tempdes fecdba9876543210 0123456789abcdef test.txt test.des

   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. RSA and SHA1 implementations should follow the same call format.

PART C – Secure Client-Server System

1. **General Description**
   You will combine your code from PartA and PartB to build a secure client-server system which provides the following: (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 sample codes and results**
   You are given 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 are also two README files that contain other important implementation details. These files define all details you need to complete the assignment, and the instructions will guide your design. **Please download the file lab4C.tar for all PARTC sample files.**

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 to the header.h file and README files provided as part of the assignment.

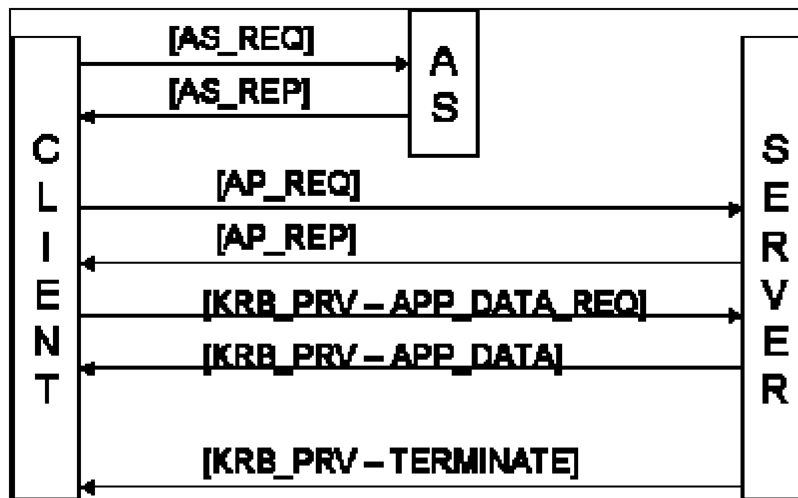

Figure #1 Functional specification

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. 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,
<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 that 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.

Deliverables:

a. For PartA: A copy of the files **client.c** and **server.c**

For PartB: A copy of the files **des.c**, **rsa.c** and **sha1.c**
For PartC: A copy of the files **client2.c**, **server2.c**, and **authserver.c**
sent to the Laboratory Assistant.

b. Printout.
   Turn in a **typed** report with the following information:
   - C code implemented for all files in parts A, B & C
   - Screenshots showing your execution and results

Note on Grading

Points will be deducted if any of the deliverables is missing.
Further, points will be deducted if the programs implemented
Do not have a name as specified above, (or)
Do not compile (or)
Do not produce the expected results. Please make sure you compare your results against the given test cases.
Do not have appropriate comments