International Institute of Information Technology, Hyderabad

Systems and Network Security (CS5470)

Lab Assignment 1: Implementation of a client-server program to access service from a server by a client (An example of Secure File Transfer Protocol)

Deadline: February 5, 2019 (Tuesday), 23:55 P.M.

Total Marks: 100 [Implementation (Coding + correct results): 75, Vice-voce: 25]

Note:- *It is strongly recommended that no student is allowed to copy programs from others. Hence, if there is any duplicate in the assignment, simply both the parties will be given zero marks without any compromisation. Rest of assignments will not be evaluated further and assignment marks will not be considered towards final grading in the course. No assignment will be taken after deadline. Name your programs as roll_no_assign_1_client.c and roll_no_assign_3_server.c for the client and server. Submit your only client.c and server.c files to TAs. If you are using other programming languages other than C, you are also welcomed.*

Description of Problem

Part 1

For establishment of a shared session key $K_{A,B}$ between the client A and the server B, you use the Diffie-Hellman key exchange protocol, which is described below:

- *Global Public Elements*
  - q: a sufficiently large prime, such that it is intractable to compute the discrete logarithms in Z_q^*.
  - α: $\alpha < q$ and α a primitive root of q.
- *User A Key Generation*
  - Select private X_A such that $X_A < q$
  - Calculate public Y_A such that $Y_A = \alpha^{X_A} \mod q$
    $A \rightarrow B : \{Y_A, q, \alpha\}$
    Here $A \rightarrow B : M$ denotes party A sends a message M to party B.
- *User B Key Generation*
  - Select private X_B such that $X_B < q$

Table 1: Encoding used in Part 2.

A = 01	K = 11	U = 21	1 = 31
B = 02	L = 12	V = 22	2 = 32
C = 03	M = 13	W = 23	3 = 33
D = 04	N = 14	X = 24	4 = 34
E = 05	O = 15	Y = 25	5 = 35
F = 06	P = 16	Z = 26	6 = 36
G = 07	Q = 17	, = 27	7 = 37
H = 08	R = 18	. = 28	8 = 38
I = 09	S = 19	? = 29	9 = 39
J = 10	T = 20	0 = 30	a = 40
b = 41	c = 41	d = 42	e = 43
f = 44	g = 45	h = 46	i = 47
j = 48	k = 49	l = 50	m = 51
n = 52	o = 53	p = 54	q = 55
r = 56	s = 57	t = 58	u = 59
v = 60	w = 61	x = 62	y = 63
z = 64	! = 65		

  - Calculate public Y_B such that $Y_B = \alpha^{X_B} \mod q$
    $B \rightarrow A : \{Y_B\}$

- *Generation of secret key by User A*
  - Compute the shared key with B as $K_{A,B} = (Y_B)^{X_A} \mod q$
- *Generation of secret key by User B*
  - Compute the shared key with A as $K_{B,A} = (Y_A)^{X_B} \mod q = K_{A,B}$

Part 2

1. When a user A wishes to create a login in the server, he/she enters his/her own id ID_A and a chosen password (maximum of TEN characters only) PW_A. User A also selects a large prime q_A using the Miller-Robin Primality test algorithm.
   Client (user) sends a message $LOGINCREAT(ID_A, PW_A, q_A)$ to the server securely encrypted using the secret key $K_{A,B}$ already established in Part 1.

   For encryption/decryption, we apply the existing generalized Caesar Cipher with the encoding technique given in Table 1. 00 indicates a space between words.

2. After receiving $LOGINCREAT$ message, the server randomly generates a salt value (known as random nonce) S_A for the user A, which is a random positive integer. The server then generates the hash value as $H(P_A||S_A||q_A)$, where $||$ is the contenation operation. The server creates an entry to the following password table for user as follows:

Table 2: Password File

User ID	*Salt*	*Hashed Password*	*Prime*
ID_{Alice}	S_{Alice}	$H(PW_{Alice}\|\|S_{Alice}\|\|q_{Alice})$	q_{Alice}
ID_{Bob}	S_{Bob}	$H(PW_{Bob}\|\|S_{Bob}\|\|q_{Bob})$	q_{Bob}
$\vdots$	$\vdots$	$\vdots$	$\vdots$
ID_{Eve}	S_{Eve}	$H(PW_{Eve}\|\|S_{Eve}\|\|q_{Eve})$	q_{Eve}

Note that before creating an entry at the server, it must check whether the ID of the requested user is already in the table. If so, reject this request, and send the $LOGINREPLY$ message with an unsuccessful/successful status to the user A securely (encrypted using the established secret key $K_{A,B}$.

You can use SHA-1 as the hash function $H(\cdot)$. Otherwise, for the sake of simplicity, you may assume that the hash function structure will be as follows: $H(x) = x^3 \bmod q_A$. More precisely, for computing the hash value $H(PW_A||S_A||q_A)$, you first calculate $PW'_A = \sum_{i=1}^{password\_length}$(ASCII value of character i) and then compute $H(PW_A||S_A||q_A)$ as
$H(PW_A||S_A||q_A) = (PW'_A + S_A + q_A)^3 \bmod q_A$.

3. In this way, assume that n users (clients) can login in the server and their corresponding ids, salt values and the hashed passwords are stored in the password file of the server B.

4. Now assume that a user, Alice wishes to access some file (say, f1.txt) stored in the server, where the file "f1.txt" will have only characters provided in Table 1. Before granting the service, the server should authenticate the user, Alice. Hence, Alice needs to enter her own id and password to the server, that is, $Alice \longrightarrow Server : AUTHREQUEST(ID_{Alice}, PW_{Alice})$.
This request is encrypted using the shared secret key $K_{A,B}$ in Part 1.

The server and client then perform the following steps:

- Server searches the user ID (login id) in the password file. If it is found, then the server selects the corresponding the salt value S_{Alice} and prime q_{Alice}.
- Server computes the hashed password $H(PW_{Alice}||S_{Alice}||q_{Alice})$.
- Server matches the computed hashed password with the stored hashed password in the password file. If they match, then the server accepts Alice as an authenticated user and grants service requested by the user (Alice).
$Server \longrightarrow Alice : AUTHREPLY(SUCCESSFUL/UNSUCCESSFUL)$.
- After receiving the $AUTHENREPLY$ and if the status is successful, then only Alice requests for a file transfer (say for f1.txt file stored in the server) with a SERVICEREQUEST message.
$Alice \longrightarrow Server : SERVICEREQUEST(ID_{Alice}, file\_name)$.
- If the file is in the server, then subsequently the server transfer the file to the client, Alice.
$Server \longrightarrow Alice : SERVICEDONE(file\_name, SUCCESSFUL/UNSUCCESSFUL)$.

Note that if the file size is more than MAX_LEN (= 1024 bytes), then the file be transmitted from the server in multiple times of maximum length MAX_LEN. In this case server will send $SERVICEDONE$ $(file\_name, SUCCESSFUL/UNSUCCESSFUL)$ muliple times to the client.

Protocol Messages to be used in implementation

Opcode	*Message*	*Description*
10	LOGINCREAT	a login create request to the server by the client
20	LOGINREPLY	an acknowledgment for successful login create request
30	AUTHREQUEST	a request for accessing service from the server
40	AUTHREPLY	successful/unsuccessful authentication from the client
50	SERVICEREQUEST	service requested to the server by a client
60	SERVICEDONE	File transfer will take place only if the file is in the server

Data structure to be used in implementation

```
#define MAX_SIZE 80
#define MAX_LEN 1024

/* Header of a message */
typedef struct {
    int opcode; /* opcode for a message */
    int s_addr; /* source address */
    int d_addr; /* destination address */
} Hdr;

/*A general message */
typedef struct {
    Hdr hdr; /* Header for a message */
    char buf[MAX_LEN]; /* Contains a plaintext message (for example some part of a transmitted file) */
    char ID[MAX_SIZE]; /* The identifier of a user */
    int q; /* A prime */
    char password[MAX_SIZE]; /* Password chosen by the user */
    int status; /* SUCCESSFUL or UNSUCCESSFUL for successful or unsuccessful result
                in LOGINREPLY and AUTHREPLY messages */
    char file[MAX_SIZE]; /* The file which will be transmitted by the server to the client */
    int dummy; /*dummy variable is used when necessary */
} Msg;
```

Instructions:

1. Write your client.c program in CLIENT directory. Run the program as: ./a.out 127.0.0.1 (for local host loop back)

2. Write your server.c program in SERVER directory. Run the program as: ./a.out