CS 5352
Project 3
Due: Nov. 23 at 11:59 pm. (No late submission allowed)

Objective – This project is designed to teach 1) development of client-server programs using a program generator *rpcgen,* 2) implementation of secure communication protocols, and 3) use of DES (data encryption standard) for protecting messages.

Problem – When a user obtains a service from a remote server, the response is generally sent over unprotected channels. Therefore messages should be encrypted to assure confidentiality and integrity of its data. It requires that a key be generated and passed to the user and server in a secured way. The user must first request for the key before contacting the server and follow a mechanism so that key could not be captured by any intruder listening the communication. The purpose of this project is to realize secured client- server communication using this mechanism.

In this mechanism, there are three components: a client program (CP), a key-server (KP), and an application server (SP). CP is owned by you, the user; SP is owned by another user, your friend, and KP is owned by an organization, which is trusted by all, like VeriSign. Assume that when a user account is created, a secret key is also created and that it is given to the user and to the key server KP by some means other than sending over the Internet; in summary, each user knows only its secret key, but KP knows the secret keys of all users. Obviously, the secret key of CP, say key (C), cannot be used for communication between CP and SP (because SP would not know it). Hence, CP must first acquire a new key to communicate with SP; it is called a *session key*. The following describes a mechanism for session key generation and its distribution.

1. CP sends a request to KP to generate a session key. This session key will allow CP to have secure communication with SP. The request contains C and S, the widely-known id of CP and SP. The request message is sent in plaintext, because there is no confidential information in it; everybody is expected to know these ID. Let {C, S} denote this request. Let curly parentheses denote a message consisting of a few items; one can view it as a 'structure' data.

2. Upon receiving this request, KP tests if the two IDs are in its key database. If so, it will generate a new session key, K_{CS}, and prepares two messages. The first message is termed token representing the following; *token* : {C, S, K_{CS}}. It is encrypted using key(S), the secret key of S, which KP should have in its key database. The encrypted token is denoted as $\{C, S, K_{CS}\}_{S.}$ The subscript S outside the curly parentheses denotes that the message is encrypted using key(S). By putting the three components together in the token, KP attests that CP is authorized to communicate with SP using the given session key.

   The second message is the reply to the request of CP and it contains four components. It is encrypted using the secret key(C) of CP and is represented by $\{C, S, K_{CS}, \text{encryptedToken}\}_{C.}$ Again, the subscript C outside the curly parentheses denotes that this message is encrypted using key(C). KP does not retain K_{CS} or any record that it processed this client request. (KP is a stateless server).

3. Upon receiving the reply, CP decrypts it using its own secret key and extracts the four components. The second component, S, attests that SP is a valid user (otherwise, its ID and secret key would not be present in KP's database) and that CP can communicate with SP using the given session key. CP cannot decrypt the token or alter it, because it does not know key(S). This completes the communication of CP with KP.

4. Let m denote a service provided by SP. m : {name, arg} is a structure. Here, name denotes name of a service and arg are the arguments. CP first encrypts m using K_{CS} to produce $\{m\}_{CS.}$ Next it prepares a request message M : {C, $\{m\}_{CS,}$ token}. It contains ID of CP in plaintext, encrypted

m, and the encrypted token received from KP. There is no need to encrypt M, since an intruder cannot determine the service requested by CP. CP now sends M to SP.

5. SP parses M and separates its three parts. Next it decrypts the token using its own key(S) and extracts the first two components of M, (i. e., C and S). SP checks that S denotes its ID, which means that the token was generated by KP for SP only. It also compares that the first component of token with the plaintext ID, C, sent by CP in M. If the two id match, SP is assured that $K_{CS.}$ was generated by KP on behalf of the same client CP who sent M. SP obtains the enclosed session key.

6. SP next uses K_{CS} and decrypts the second part of M and obtains the service requested by CP in plaintext. SP processes the request, computes the response, say r, prepares a reply message, {C, S, r}. SP next encrypts the reply using the session key K_{CS} to create R: $\{C, S, r\}_{CS}$ and sends R to CP.

7. CP is expecting a reply from SP only. It uses the session key K_{CS} to decrypt R and compares its first two components with the IDs it sent. If they match, CP is assured that the message is in response to its previous request. It uses the response r gladly. This is the end of the story of how to implement secure client-server programs. Hurray!

The above should be seen as a two part solution – acquisition of a session key (steps 1-3), and delivery of authenticated service (steps 4-7). CP and SP can use the same session key for any number of service calls (in reality, session keys have a limited life and are used for a specified period only).

Digesting the solution - In order to understand above mechanism clearly, it is strongly suggested that you first convert above description into a diagram with three vertices, CP, KP, and SP.
(a) Then define messages exchanged between CP and KP and between CP and SP as structures; note that there are six different messages all together.
(b) Give the pseudo code of the three procedures,
one for each component.
(c) Fill each procedure with statements such as prepare message X, encrypt it using key V, perform test Z, send message X to U and/or receive message X from U, etc.
The objective of this step is to essentially rewrite above solution tracing flow of program structures. If you did not do this part well in advance of the due date of the project, you will be hurting feelings of your instructor.

Implementation – This project requires you to develop three program files, CP, KP, and SP. Each program performs essentially two *core* tasks:
(1) Compose request messages and/or parse reply messages.
(2) Send a request message and receive the reply message.
The code for the second task can be generated automatically using a program, rpcgen, on Solaris, i.e., you will not have to write any send() and receive() code. Therefore, the project development is divided into two parts as follows:

A. Develop skeleton of the three programs; each composes a dummy request and parses a dummy response; use automatically generated code to exchange messages.
B. Fill dummy request and response messages with appropriate data as per the above solution and use cryptography.

Create a folder Proj3 in your cs5352 folder and two subfolders Proj3A and Proj3B in it. As in previous projects, development of this project is also divided into a few stages and each stage is further broken into steps. **It is absolutely important that you implement each step and test it before you proceed further.**

Stage0: A number of files have been created to help you to understand the problem, provide necessary code and test data. Make a note of all what is given.

(a) Make a hard copy of the class notes posted on the class web site on client-server programming and its automatic generation. Make note of the sentences written in red ink. Next, copy a number of tar files from "~cs5352/projects/proj3" to your proj3 folder. Each tar file contains a readme to guide you.

(b) "rpc.tar" contains a small client-server program to show you how to use rpcgen. Study the two C files and the Makefile. Make a note of names of files generated by rpcgen. Experiment with the given program as described in the readme file. client.c and proc.c print a number of debugging message. Make sure that your server, generated by Makefile, also prints the bugging messages. If not, see me. Use of Makefile is described in a note posted in the UNIX-C section.

Before you start coding this project, it is important that you fully understand the interface definition coding. Given proc.c provides three services. Suppose that you want the server to provide one more service, say ADD; given two integers, it computes the sum and returns it as the reply. Expand the given files CS.x, client.c and proc.c, to incorporate this service. Run and test the generated client-server programs.

Add print statements in proc.c to print the arguments and reply of procedures. Build the server again to test if your server prints any output. First start the server in background, 'SERVER & <enter>'; If the server does not print any output on the terminal, kill it and start it the foreground, '<SERVER <enter>'. If it still does not print, it means that the automatically generated server has closed the standard output. The only option left is then to save the output to a file. For this purpose, open a file, FILE *fin=fopen(….), in the append mode at the beginning of each procedure, change 'printf()', to fprintf(), and close the file at the end of the procedure. View this file for output. It is very important that you test this step before continuing.

(c) crypt.tar contains small programs to show use of DES for encrypting and decrypting plaintext data. It also contains a procedure to generate large random integers, which you can use to compute the session key. This file is needed for partB.

(d) DB.tar contains a readme file, a database DB.key, and two key files, Ckey.h and Skey.h. DB.key is the database of secrete keys to be used by KP. The data is coded as a single long string, which should be viewed as a table of records; each record contains an 8-byte user-id and an 8-byte secret key of that user. KP reads 16 bytes at a time of this file. The two header files are to be included in the CP and SP programs. Each contains the secret key of the individual. Examine the two header files and you should see that these are included in DB. **Your project will be tested with these databases only; do not create your own.**

(e) testData contains data to test this project.

(f) ~cs5352/projects/proj3/" contains executable of this project developed by an ex- undergrad student. After you have understood a stage, run his program for that stage to see the output your program is expected to generate.

Part A: Development of CP, KP, and SP programs:

1. Implement the first half (steps 1-3) by developing client program C1P and KP,
2. Implement the second half (steps 4-7) by developing client program C2P and SP, and
3. Glue C1P and C2P into a single client program CP, which would communicate with both KP and SP.

Stage A1: Develop skeleton of CP1 and KP programs:

Step 1 – Create a folder A1, and develop an interface file, C1K.x. Define data type of the request and the reply messages (shared by C1P and KP) as per the following details. Assume that IDs C and S are 8 byte long plaintext data and that the session key is also an 8 byte long random text. Therefore, the token can be defined as a structure containing fields for {C, S, K_{CS}}; this structure can be also viewed as a 24 byte array, with C, S, and K_{CS} stored in the first, second, and third 8-byte segments, respectively. Therefore, there are two ways to code the token structure:

```
struct Token {
        unsigned char C[8],
            unsigned char S[8],
            unsigned key[8]
};
or
struct Token {
            unsigned char T[24]
};
```

The second coding approach requires C, S, and session key be copied at right addresses, e.g., Token, Token+8, and Token+16, respectively. Use memcpy. Note that the "unsigned char" is not a data type in the language compiled by rpcgen. Hence, it must be included in a struct. In that case, it leaves translation for the C compiler.

Likewise, the reply, {C, S, K_{CS}, token}, can be declared in two ways, either as a structure of four items, or as a structure of a single 48 byte array.

In network communication, sending the length of the messages is common. Hence, each request and reply message should be defined as a higher level structure containing two fields, length and message structure. For example,

```
struct netReply {
            int replyLen;
            struct reply R;
}
```

The sender not only fills the message structure as per its use, but it also fills the length field. However, in case of errors, the sender sets the length field to a negative integer and either leaves the reply R unfilled or stored an appropriate error message.

It is important that you follow above details strictly when coding C1K.x. For example, if K_{CS} is 8-bytes long, it should be declared as an unsigned character array of size 8 (and <u>not</u> 9 to allow space for '\0'). If you want to print it, use the following; define char junk[9]; junk[8]='\0'; when you want to print K_{CS}, use memcpy ( ) to copy it to junk[] and then print junk, instead of K_{CS} directly. Do not use string functions on the fields of message structures since such fields do not contain the string terminating null character. You can use string functions on internal variables.

Note that the interface file is used to define only the arguments and name of service procedures. You should not define other variables or internal procedures, local to the server programs in the interface file. You can always create a separate header file and include it in client.c or proc.c.

If you decide to copy given CS.x to create your C1K.x, instead of coding your own from scratch, at least change the name such as AS_PROG to CK_PROG. That's creativity!

Step 2 – Customize the given Makefile by renaming program names with your client and server programs. Since you would have to merge two Makefiles in stage A3, it is advised that you use distinct names so that there is no problem in merging two Makefiles. For example, rename 'BIN' as BIN_C1K, 'GEN' as 'GEN_C1K', SERVER as SERVER_C1K, etc.

Step 3 – Edit the given client.c to develop your C1P.c. Use memcpy( ) to copy fields of the request/replay structures.

C1P is called with three arguments: 1) user-id C, 2) user-id S, and 3) hostname of the key server KP; C and S denote user chosen random text in this part. If the chosen text is shorter than 8 bytes, CP1 pads it with blanks. To implement this piece, fill 8-byte area with null characters using memset() and then copy the user provided text using memcpy(); a shorter input will be automatically padded with nulls.

Step 4 – The server KP contains two parts: (1) automatically generated main ( ), called the server stub, and (2) service procedures coded in KP_proc.c. Edit proc.c to create your KP_proc.c. In this part, it contains just one service procedure, say requestSessionKey( ) and a few local procedures. C1P calls this service procedure passing {C, S} either as a structure containing two items, or just as a single array of 16 bytes.

requestSessionKey() will call some internal procedures before generating a session key. These are: KP_validator(), keyGen( ), token_builder(), and finally replyBuilder(). Since purpose of this stage is to build only a skeleton of KP in this part, validator() always returns true, (i.e., it does not test if C and S are in its database). keyGen( ) also does not generate any real session key; it just copies a pre-defined, 8-byte random character string into a field allocated to $K_{CS.}$ tokenBuilder( ) uses a few memcpy() to prepare the token as a 24 byte string, or as the structure you decided upon in earlier steps. replyBuilder() uses memcpy() to compose the reply as a single item 48 byte long byte array, or as a structure again as you decided before. Finally, requestSessionKey () returns the reply. It is automatically sent by the server stub to CP.

It is important that you use suggested names for source files. You should therefore code C1P.c, KP_proc.c, and C1K.x. Name the client program C1P and the server program KEYSERVER in the Makefile. It will allow the grader to test your project by running a script.

Step 5 - Testing – Run CP1 and KP in two separate windows. C1P and KP should print components of the request and reply messages to verify that each message segment is transferred to the other side correctly. Print statement should produce suitably labeled output so that the grader can easily identify all pieces of data. (e.g., “KeyServer: Received message ….”.)

Stage A2: Develop the skeleton of CP2 and SP.

Step 6 – Create another folder A2 and copy files from “rpc.tar”. Develop C2S.x. Before you develop SP, you will need a slightly different understanding of problem. Since SP can be any application server, it should be designed to provide multiple services using a separate procedure for each service. Therefore a message, denoted by ‘m’ in the problem description, really represents the name of a service its arguments. Therefore, sending the message {C, $\{m\}_{CS,}$ token } by CP in our problem description can be viewed as making a remote procedure call: M (C, $\{arg\}_{CS,}$ token); where M is the name of the service, and args is the argument. For example, to compute “ADD 5, 7” using a remote service procedure, the client would call “ADD(C, $\{5,7\}_{CS}$, token)”. In other words, in our implementation, only the arguments are encrypted. This change in representation is made here to simplify implementation of SP. Note that Encryption is dealt in partB. Hence, “M(C, {args}, token)” is the right representation for partA. Here, {args} denotes a structure containing arguments as defined in C2S.x

In this project, SP provides two very similar services: 1) ALPHA(s) and 2) NUMERIC(s), where ‘s’ is a character string. ALPHA( ) drops all non-letter characters from ‘s’ and returns the remaining string; for example, if ‘s’ is “G*8j”, it should return “Gj”. NUMERIC () does almost the same except that it keeps only digits. (The reason for choosing two very similar services is to reduce coding efforts.)

Step 7 – Edit the given proc.c to derive SP_proc.c for this part. In addition to two service procedures, SP_proc.c also has some of its own internal procedures. For example, SP_validator ( ) parses the token and validates the sender. In this stage, it does do anything and always returns true. Another internal procedure, replyBuilder() builds the responses to be sent to CP. It returns the response and it is automatically sent to CP by the server stub.

Step 8 – Edit the given client.c to develop C2P. Do NOT use the names you used in C1P.c for local variables, because you will glue these two files in the next stage to create a unified CP.c. The token does not play any role in partA. C2P defines a random text of 24 bytes and uses it as a token, as if it obtained it from KP. C2P calls ALPHA(C, s, token) and NUMERIC (C, s, token) once each. Therefore, the argument for each of the two services is the same structure containing argument string and its length. This structure can be used for reply also.

C2P then prompts the user for the argument of each service separately, first for ALPHA and then for NUMERIC, as if they are very different services.

Make sure that you use different names in C2S.x than used in C1K.x of stageA1, i.e., if the KP server is declared as KP-PROG, it is something like SP_PROG in C2S.x. Use different program numbers for the two servers as well. C2P uses the same three arguments, C, S, and hostname of SP server host.

It is important that you use suggested names for source files. You should therefore code C2P.c, SP_proc.c, and C2S.x for this stage. Name the client program C2P and the server program SERVER in the Makefile. By following this requirement, you will simplify grading of the project.

Step 9 -Testing – Print all segments of request and reply messages as in the previous stage in the client and server programs and compare them.

Debugging: Follow instructions given in the Readme file for combining the client and service proc file. It should help you to debug the two programs as a single program.

Stage A3: Combining client programs.

Step 10 – Create a folder 'A3'. Copy source and Makefile from A1 and A2. The purpose of this stage is to produce combined client program, CP. CP will first obtain service from KP (the session key and the token) and then from SP (reply from ALPHA/NUMERIC). Hence, to create a combined client program, append CP2.c to C1P.c to create CP.c. Strip off the second main(). Then add memcpy( ) to copy the token received from KP in C1P to the storage used in C2P for the token sent to SP. Also update argv[] of CP so that it has four arguments - IDs of C and S, and the hostnames of KP and SP servers.
No other changes are needed to create CP.c. No changes are needed at all in the KP and SP programs here.

It is important that you change variables such as AS_PROG defined in the given interface file to something like CK_PROG or CS_PROG; otherwise, when you include files CK.h and CS.h to CP.c, it will cause problems.

Step – 11 - Combine two Makefiles. You should have essentially two copies of each line of the given Makefile such as BIN_CK and BIN_CS. Add BIN_ALL = SERVER_CK SERVER_CS CLIENT_ALL. Define a target CLIENT_ALL so that CP is compiled with the two xdr and two client stubs files.

Step 12 – Testing - Run CP, KP, and SP in three separate windows. Start KP and SP first and then CP. SP should print the same token value printed by KP.

Stage B1: Introducing cryptography

Copy files of Proj3A/A3 folder to Proj3B/B1 and expand them as described below. There are two objectives in this stage – exchange encrypted messages, and validate the sender.

Step 13 – Follow the readme file from "crypt.tar", run the given crypt program, des.c, on plaintext input of different size such as 8, 16, and 24 bytes, and note down the size of the encrypted text. You will see that the size increases non-linearly.

In partA, all components exchanged plaintext messages, since each message was viewed as a byte array of text characters. The purpose of this stage is to develop a strategy so that components exchange encrypted messages but something simple enough that you do not have to make any substantial modification to existing code. Since encryption/decryption is not part of the logic of the service procedures, our approach should add encryption/decryption code as outside logic. The following describes our approach.

1. Add two new structures, say enc_request and enc_reply, in the interface definition file. Each contain two fields, say enc_Len and enc_message_array; the first field holds length and second is a byte array of suitable length.

2. Revise the prototype of the service procedures to use enc_request and enc_reply in place of the original request and reply structure in the interface definition file and service code file.
3. Prepare the request and reply messages in previously declared data structures as before. //No change in the Stage A3 code.

4. Copy request and reply structure in a locally defined text array of arbitrary length using memcpy(). See ~cs5352/tutorials/struct_Array_conversion.c for example.

5. Encrypt the plaintext array and save the encrypted text output in enc_message_array.

6. Store length of encrypted text in enc_Len.

7. Use procedure calls as before; enc_request and enc_reply are transmitted now.

8. SP decrypts the argument into a locally defined text array.

9. SP copies the text array to locally defined variable of original request structure using memcpy().

10. SP uses existing code to process and compute reply; your stage A3 code.

11. Follow above logic to encrypt and send enc_reply.

Step 14 – Include the given header file CKey.h in CP.c. Expand CP program as per above logic. After decrypting the reply of the key server, KP, it extracts the third 8-byte segment of the reply to obtain the session key $K_{CS.}$

Do not forget to revise print statements. Do not print encrypted data like you would print ordinary ascii text. Print each byte as an integer; see the given crypt program. When you print encrypted data, some non-printable data may change terminal settings, causing the terminal not to work. I will be unable to grade your programs (you would not get any credit), if your program disables my terminal. **Hence follow this requirement.**

Step 15 - Expand the internal procedures called by requestSessionKey( ) of KP as follows.

(1) KP_Validator() opens the given key database file, DB.key, reads every 16-byte data into an buffer and compares the given user id C (or S) with the first 8-byte to check if they match with some stored id, and returns the corresponding secret key. This procedure is called twice to validate C and S individually.

(2) Copy the given code randKey() from the crypt program into keyGen() to generate the session key $K_{CS.}$

(3) Complete tokenBuilder() to compose the token and encrypt it using the secret key of S.

(4) Complete replyBuilder() to compose the *token* and encrypt it using it using the secret key of S. (You should examine that the secrete keys given to the users via files CKey.h and SKey.h match with the keys stored in the database file.) **Use the given key database and not create your own.**

Step 16 - Include Skey.h file to SP_ proc.c program. Complete SP_validator ( ); it should decrypt the token using its secret key, Skey, extract K_{CS}, and then compare the id sent by CP with the one stored in the token.

Step 17 - Expand service procedures of SP; each procedure should decrypt the argument m using the session key extracted from the token, compute the output, call replyBuilder() to compose the reply and encrypt it. The service procedure returns the reply to the server stub.

Step 18 – Test – Repeat step 12 to test this stage. Follow the given “testData” file.

Stage B2: - Simulating threats from attackers - While designing previous stages, we assumed that all components work correctly and that each component sends/receives valid messages. In this situation, senders of messages set the length field of the request/reply messages to positive integers to denote the true length of the messages. The purpose of this stage is to consider various threats and determine suitable responses. In case authorization and validation tests fail, because of the reasons given below, servers set the length field to a negative integer to indicate failure. The receiver of a message always first checks the length field and if it is negative, it discards the message and prints an appropriate error message.

Since calls from CP to KP or SP are sent over public network, it is possible for someone, an intruder, to listen to the communication and send bogus message to the waiting party pretending either as CP, KP, or SP. We assume that intruders are civilized that they follow protocols described in this project, i.e., they construct fabricated request/reply but use the mechanism implemented in this project. This stage requires you to fix your code to handle failures.

1. The attacker sitting in a router intercepts the request of CP to KP, changes the id of C or S in the request, and forwards the request. In this case, sessionKey( ) should fail to find the affected id in the given database. KP should set the reply length to -1.
2. The attacker intercepts the requests and sends a bogus reply before KP is able to respond to the CP's request. CP would process the first reply only. Since the attacker is not expected to know the secret key of CP, it would encrypt the reply using some random key. CP should find that the id C and/or S stored in the reply does not match with what it sent to K.
3. Assume that the attacker replaces the token part of a genuine request from CP to SP by a bogus token. In this case, SP should find out of the attack and send a reply of length -1.
4. The attacker intercepts a service requests and sends a bogus reply to CP before SP is able to respond. CP should detect this case and reject the response.

Create a folder B2 in proj3B and copy all source files of B1. Modify the CP.c, SP.c and KP.c slightly as described below to test above four cases. For example for case 3, alter CP.c so that instead of using the token received from KP, it copies garbage token when framing service requests to SP. It requires changing just one statement in CP.c. Run make and rename the binary of the client to I-CP (to denote intruder CP). You should develop and I-KP and I-SP for cases 2 and 4 by making small changes in KP.c and SP.c.
To test case 1, use the original CP and KP, but some C or S not in DB. For case 2, use original CP and K-IP. For case 3, use I-CP and original SP. For case 4, use original CP and I-SP. In each case, the Component, which detects a problem should print appropriate message.

Submission: Before submitting Proj3 folder, clear all binaries files and generated files. The grader will generate them using your Makefiles. Copy the testData file in both proj3A and proj3B. Also leave a readme. Use suggested filenames so that the grader would have no difficulty recognizing your code files. Part A is worth 65% and part B 35%.

Please remove core files before submitting the project. A core is a dump of entire process space. On Solaris, it is as large as 32MB. I do not need such files.