Assignment 4

Handed Out: November 19, 1999 Due: December 10, 1999

Multi-threaded Secure Login Across The Internet (50 Marks)

In the early days of the Internet people used programs such as "telnet" to login to remote machines across networks. As networks stopped being internal to single companies or institutions, security became an issue. One problem is that of "packet sniffing": as packets travel across the Internet, often through "untrusted" machines, unscrupulous 3rd parties may examine them. If one such packet contains your password (as required by the remote system you are logging in to, then security on that machine is breached when the eavesdropper learns your login name and password! Also, the information you transmit back and forth may be sensitive, and may also be observed by eavesdroppers. Any un-encrypted communication you send across the Internet is like writing your secrets on a postcard and mailing it: anyone can read it. Breaches do happen. In the past 2 years CSlab and CDF at the University of Toronto have detected many security breaches based on passwords being "sniffed."

To combat this problem, many schemes for encrypting login sessions across the Internet have been devised. One popular scheme, which some of you will be familiar with, and which all of you *should* be familiar with, is called SSH. In this assignment, you will write a simplified client/server that behaves in a manner similar to SSH.

The Protocol

We require a "protocol" so that the clients and servers you write can operate with one another without problem (throughout the course of this assignment, I urge you to test your client against other's servers and vice versa). A "protocol" is just a set of rules about how the two programs will communicate.

The first thing the server does upon receiving a new connection is to transmit a sequence of 128 characters that will form an "encryption key" (this corresponds to 1024 bits of encryption). This means that anyone who sniffs the first 128 characters of our transmission and who knows we are using RC4 encryption could easily crack our system, so it is not "truly secure." In the interest of keeping this assignment simple (*i.e.* no public-private key exchange is required) we won't worry about this. The server may generate these 128 bytes any way it wishes, but it must generate a different string every time a new connection is made.

All communications between the client and server after the initial 128 byes are encrypted.

Connecting the Remote Login

After receiving the encryption key, the client transmits the following information to the server:

```
login <username>\n
```

The server will parse the two tokens to determine that this is a login request, and will proceed to create a login for the specified user on the machine the server is running on. If this cannot be done, an encrypted error message is returned and the connection terminated.

Note: do not transmit a null-character (0) at the end of the string … transmit characters only up to and including the new-line (`\n`).

Port Forwarding

Port forwarding allows other non-secure clients to access services on remote machines in a secure manner. For example, a

POP3 mail client connects to port 110 on a remote machine, but that connection is insecure so an eavesdropper can read the data transmitted. If instead the POP3 server connected to some port on the local machine that was (securely) forwarded to port 110 of the machine on which the secure server is running, then data transmitted is encrypted and can't be sniffed. (Since POP3 clients have to transmit a password to the POP3 server this is a good idea!)

After receiving the encryption key, the client transmits the following information to the server:

```
forward <portnum> <remoteHost>\n
```

The server will parse the three tokens to determine that this is a port-forwarding request, and will forward all data received on this connection to port <`portnum`> on the machine specified by <`remoteHost`>. The parameter <`portnum`> is an ASCII representation of the port number to forward, so for example if we wish to forward port 6312, <`portnum`> is the character '6' followed by '3' followed by '1' followed by '2'.

After the server has received the request, all subsequent bytes are forwarded to the desired port. In the event that the forwarded connection cannot be made, an encrypted error message should be returned.

Note: do not transmit a null-character (0) at the end of the string … transmit characters only up to and including the new-line (`\n`).

The Client

The synopsis for the client is as follows:

```
mySC <username> <hostname> <port> L<lport1>:<host1>:R<rport1> L<lport1>:<host1>:R<rport1> ...
```

The command line parameters are used as follows:

`<username>`	the login name for the user on the remote system
`<hostname>`	the name of the remote system to login to
`<port>`	the port on which the server (mySS) is listening
`<lportN>`	the Nth local port to forward
`<rportN>`	the Nth remote port to forward to
`<hostN>`	the Nth remote host to forward to

The first two are mandatory; and there may be any number (including 0) of forwarded ports. An example of using this is as follows:

mySC maclean eddie.cdf L1234:mail.cs.toronto.edu:R110

This will create a remote login session for user maclean on eddie.cdf, and forward any attempts to connect to port 1234 on the local machine to port 110 on `mail.cs.toronto.edu`.

Your client should do the following:

1. For each forwarded port, spawn a thread that listens for **local connections only** on that port. Each time a connection is made, spawn a thread to forward that connection to the server, encrypting all data received before transmitting it to the

2. Once the port-forwarding threads are spawned

- create a connection to the secure server for a remote login
- save the current terminal state (see **libRawTerm.a** below)
- put the terminal into "raw" mode (see **libRawTerm.a** below)
- using `select()`, monitor the keyboard and the socket connection for new data coming in; when a new character is

available from the keyboard, encrypt it and write it to the socket; when a new character is available from the socket, read it and decrypt it and write it to stdout (the display).

- when the connection is terminated, restore the terminal to its original state (see **libRawTerm.a** below), and exit(). [**Note:** this will kill all threads handling forwarded connections … don't worry about it]

- all of the remote login code may take place in `main()` if you wish

The Server

The synopsis for the server is as follows:

```
mySS
```

The server, once invoked, listens for connections on a port whose number is calculated as follows:

```
portNum = 4000 + getuid()
```

The `getuid()` function returns the user ID of the person running mySS. Since every user has a unique user ID, this ensures that no two CSC209 students will attempt to bind the same port, even if they run `mySS` on the same machine.

The server works as follows:

1. Set up a listening socket on the appropriate port

2. As each new connection comes in, compute and transmit the 128-byte key.

3. Read the request header from the client, and spawn the appropriate thread to handle it

4. For a remote login request, the thread should create a pipe, spawn a child via `fork()` and `exec()` a copy of `rlogin` where `stdin`, `stdout` and `stderr` have been redirected into the pipe via a call to `dup2()`. As the thread receives characters from the client (via the socket), it decrypts them and writes them to the pipe. As the thread reads characters from the pipe, it encrypts them and writes them to the socket.

5. For a port-forwarding request, the thread should open a connection to the requested port on the requested host (called the 'remote connection'). As each character is received from the client, decrypt it and write it to the remote connection. As each byte is received from the remote connection, encrypt it and send it to the client.

6. Every time a new connection is made, the server prints (to `stdout`), the time of the new connection, its type (login/forward), the client who connected (IP address of client, and in the case of forwarding which remote host and port are being connected to)

A Note on Encryption

You are going to be encrypting data going both ways. Since the order of bytes received/transmitted by the client may be different than the order of those same bytes as seen by the server, we must use two encryption "channels", one for each direction. This means you will create two encryption keys, one for each direction.

Code I Will Provide For You

I will provide two pieces of code to make this assignment simpler. All files will be found in `/u/csc209h/include` and `/u/csc209h/lib`.

1. **`libRC4.a, rc4.h`:** this header file together with its library will allow you to perform RC4 encryption/decryption of a character stream. In `/u/csc209h/src` I will provide the file TestRC4.c which will demonstrate the use of this library.

2. **`libRawTerm.a, RawTerm.h`:** this header file and library will allow you to place the client's terminal into "raw" mode. In this mode no processing is done before characters reach the client, meaning that the client will be able transmit all keys pressed directly across the network connection. For example, if the client's user types ^C, then instead of sending SIGINT to the client, the ^C is transmitted to the remote login session, and SIGINT is sent to the remote process.

I will also provide source code for each of these libraries, but you are not to compile it directly into your code! Doing so will result in a 5-mark penalty for each of the client and server. The source is only provided so that you may see what these libraries are doing, and so those students writing their assignments at home under LINUX can use them before porting their code to CDF.

Some Requirements

- Use at least 10 `assert()` statements.
- You must use `select()`; marks will be deducted for improper use (if in doubt, ask).
- You may only use `fork()`/`exec()` for invoking `rlogin`; all other concurrency must be achieved using POSIX threads.
- You must use multiple files, and provide a makefile, which compiles both your client and server programs.
- You may only use INET domain sockets.
- Your code must work interchangeably with a client and server that I will provide no later than November 25th. That is, your client must work with my server, and your server must work with my client

Helpful Stuff

1. To create a remote login for the client, have the server thread managing the communications with the client `fork()` a child, and after redirecting `stdin`, `stdout` and `stderr` make the following call:
```
if (strcmp(username,me) == 0)
  execlp("rlogin", "rlogin", "-l", "", "localhost", (char *)0);
else
  execlp("rlogin", "rlogin", "-l", username, "localhost", (char *)0);
```
where `username` points to a null-terminated string containing the login name to be used, and `me` points to a null-terminated string containing the login name of the person running the server. The importance of this statement will be explained in class.

2. To make the server port available for re-use immediately upon termination of the server, include the following code fragment in your server before binding the socket:
```
int opt = 1 ;
if (setsockopt(soc, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)))
{
  perror("Unable to set server socket for re-use! "); exit(1);
}
```
where `soc` is the socket your server will be listening on.

3. Likewise, when you write data to a socket, don't write null characters that are not actually part of the data.

4. **Don't try and write the whole program at once** (remember, Rome wasn't built in a day). Instead, break it down into manageable chunks, and code/test each chunk in some order. A sample breakdown might be:
   * write remote login connection without encryption
   * add encryption to above
   * write port forwarding, without encryption
   * add encryption to above

Of course, each of these components can be broken down into smaller subtasks. It's better to hand in a working but incomplete program than a complete program that doesn't work at all.

5. To link the libraries, include the following parameters to gcc:
   `-L/u/csc209h/lib –lRawTerm –lRC4`

6. Comment out the following line in your `.logout` file (*i.e.* place a '#' in front of it):
   `/usr/bin/kill –HUP –1 >& /dev/null`

Submitting A4

When you are ready to submit your programs, you will use `tar` to combine all your source files (`*.c` and `*.h`) and your makefile into a single file named `secure.tar`. Use `gzip` to compress this file, thus creating `secure.tar.gz`.

You must hand in a printed version of your program, as well as submitting it electronically on CDF using "**`submit -N a4 csc209h secure.tar.gz`**". You can overwrite a previous submission by adding the "**`-f`**" switch to the submit command. No external documentation is required, but your program should be well documented.

The assignment will be marked with 25 marks for each of the client and server. For each part of the assignment 8 marks will be allocated for style, and 17 for correct operation.

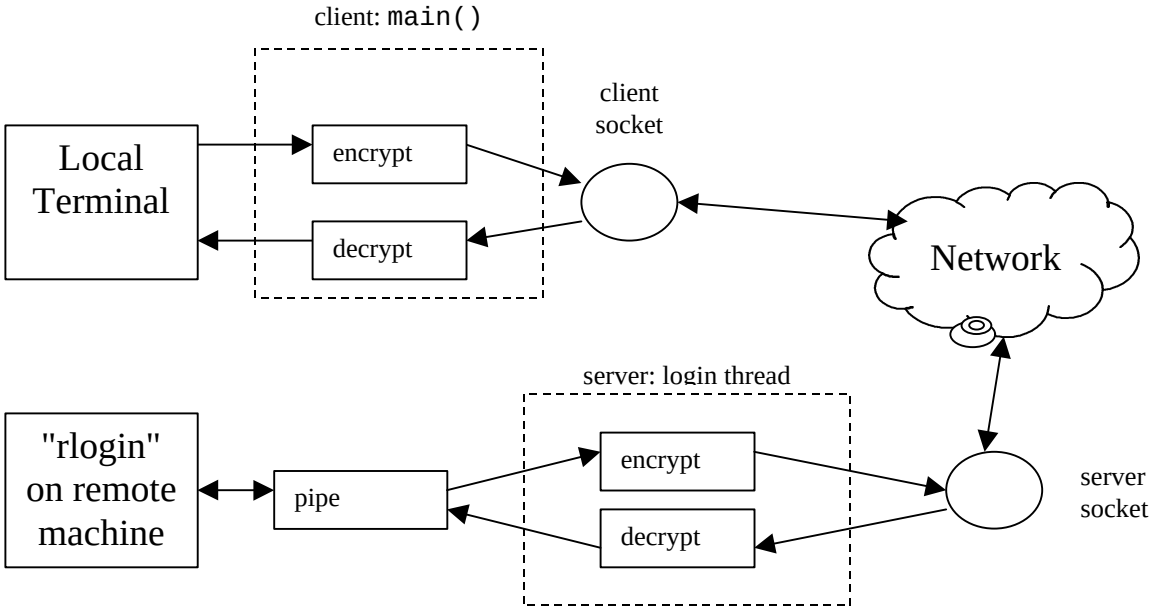
If you cannot complete the assignment: Don't panic. Marks will be given for partial completion. Try to achieve one of the following sub-goals:

1. remote login with no port forwarding or encryption (client has no threading, server has one thread for each remote login that does nothing once connection is set up except wait for child [rlogin process] to terminate)

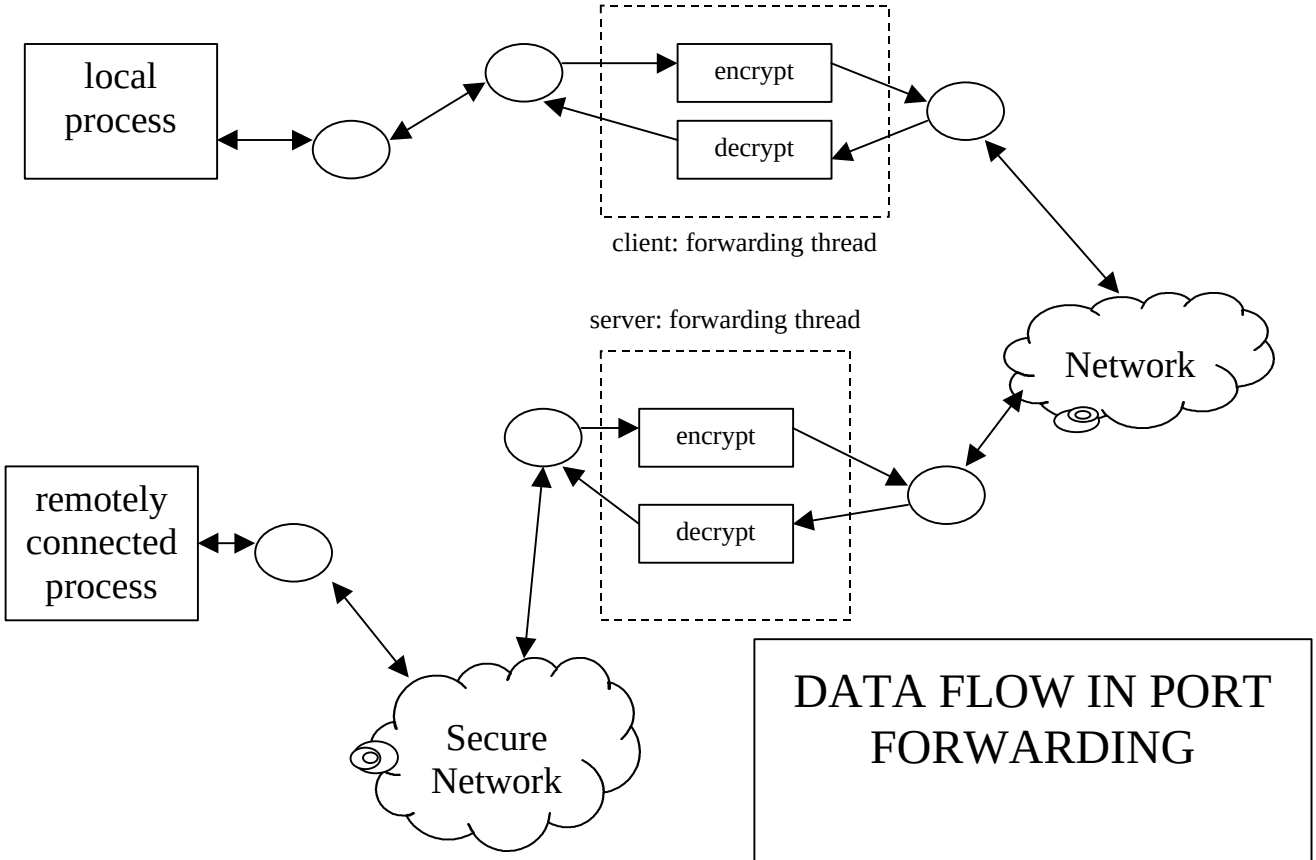
2. remote login with no port forwarding, encryption works (client has no threading; server has one thread for each remote login that does encryption/decryption on the data stream and waits for child [rlogin process]to terminate)

3. encrypted remote login with non-encrypted port forwarding (client has one thread for each port forwarded, plus one thread for each actual forwarded connection; server has one thread for each remote login [see above] plus one thread for each actual forwarded connection—these threads do nothing but read data from the client side and write them to the remote connection, and *vice versa*)

4. same as the previous item, but now the port forwarding supports encryption



DATA FLOW IN A REMOTE LOGIN



DATA FLOW IN PORT FORWARDING