Assignment 4

Handed Out: March 9, 2000 Due: April 7, 2000

Multi-threaded Secure FTP Client/Server (50 Marks)

One of the great advantages of living in a networked world is the ability to move information (files) quickly and cheaply from one place to another. For example, if you need a new driver for your video card, just download it from the manufacturer's ftp/web site (you used to have to call up and order disks with your credit card ☹). The `ftp` program is one of the original (and still most popular) utilities for transferring files across a network.

In today's networked world, security is always a concern. When you transfer files with the normal ftp client/server, the data is transmitted unencrypted, so anyone on the network can potentially see your files' contents! It would be nice to have an encrypted version of FTP.

For this assignment we will write a pair of (greatly) simplified encrypted ftp client and server programs. We will use RC4 encryption (one of the ciphers used in commercial software such as Netscape Navigator), and learn about how UNIX does password authentication.

The Protocol

We require a "protocol" so that the client and server 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 random 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. Notice that all messages between the client and server terminate with '`\n`', so the easiest way to read a message is one character at a time until '`\n`' is received.

User Authentication

Before allowing a connected client to do anything, it must be authenticated. A user name and password are transmitted from the client to the server, and the server authenticates these. If authentication fails, the server disconnects immediately. Otherwise, it allows further processing of commands. Authentication is achieved as follows:

1. The client sends a string of the form "`auth <username> <password>\n`" to the server (**Note:** the `NULL` character at the end of strings is **not** transmitted).

2. The server opens a local file `authenticate.ftp` and searches for the name `<username>`. If found, it uses the UNIX system call `crypt()` to compare `<password>` with the corresponding password from the authentication file.

3. If successful, the server returns "`auth OK\n`" and the programs continue.

4. If unsuccessful, the server returns "`auth FAILED\n`" and terminates the connection with the client.

Authentication File

The authentication file, `authenticate.ftp`, is a text file where each line contains authentication information of the form

`<username> <encryptedPassword>`

It is assumed that `<username>` contains only the characters `a-z`, `A-Z` and `0-9`. Lines are terminated with a single '`\n`' character. The `<encryptedPassword>` is the result of passing some plain-text password through the UNIX system call `crypt()`. I will provide an example of doing this.

File Transfer Client ⇒ Server

When the user wishes to transfer a file to the remote machine, s/he types "`put <filename>`". The following exchange then takes place:

1. The client sends to the server "`put <fileName> <fileSize>\n`" where `<fileSize>` is an ASCII representation of the file's size in bytes.

2. The server prepares the destination file for writing and responds with "`put READY\n`".

3. The client, upon receiving acknowledgement from the server, sends the file data.

4. When the server has received `<fileSize>` bytes worth of data, it replies with "`put OK\n`".

5. If the server is unable to accept the file, it replies to step 1 with "`put FAILED\n`". If the server is able to accept the file, but unable to write all the data in to the file, it should still read all of the data, and then respond with "`put FAILED\n`".

File Transfer Server ⇒ Client

When the user wishes to transfer a file from the remote machine, s/he types "`get <filename>`". The following exchange then takes place:

1. The client first opens the desired filename for writing, and then sends to the server "`get <fileName>\n`"

2. The server prepares the destination file for writing and responds with "`get READY <fileSize>\n`" where `<fileSize>` is an ASCII representation of the file's size in bytes.

3. When the server then proceeds to send `<fileSize>` bytes worth of data. The client replies with "`get OK\n`" (regardless of whether all the data was written to the destination file - it only reports this error to the user).

4. If the server is unable to send the file, it replies to step 1 with "`get FAILED\n`".

Note: If you write a function for opening a file and reading/writing it from/to a socket, it can be used in both the client and the server. All files are transferred in 'binary mode', which means neither the client nor server attempts to interpret the data, they just move it back and forth.

Directory Listing

It is necessary to allow the user to list the available files. The user types "`dirlist`" to the client, and the following exchange takes place:

1. The client sends the string "`dirlist\n`" to the server.

2. The server replies with "`dirlist OK\n`" followed by "`<DirName>\n`" where `<DirName>` is the absolute pathname of the current ftp directory of the thread managing the connection. The server then sends the following for each file: "`<fileName> <fileSize> <LastModifiedDate> <Filetype> <Readable>\n`". The first three items are self explanatory. `<fileType>` is one of "`R`" (for 'regular'), "`D`" (for 'directory') or "`O`" (for 'other'). `<Readable>` is "`readable`" if the file is readable by the server process, or "`not-readable`" otherwise. When the last file has been sent, the server sends an additional "`\n`".

3. If for any reason the server is unable to send the information, it returns "`dirlist FAILED\n`".

4. The client displays the information returned by the server until two consecutive '`\n`' characters are received.

Changing Directory

When the client first connects, the server thread starts in the directory from which the server was run. If the client wishes to transfer files from another directory, they must first change the current directory for their connection. When the user wishes to change his/her working ftp directory s/he types "`cd <newDir>`" to the client program. The following exchange takes place between the client and server:

1. The client send "`cd <newDir>\n`" to the server.

2. The server thread handling the connection attempts to open the new directory for reading. If successful it `returns "cd OK\n`", otherwise it returns "`cd FAILED\n`".

Note that using the command `chdir()` will change the current working directory for all threads, so each thread must instead maintain it's own record of its current directory.

The Client

The synopsis for the client is as follows:

```
sFtpC <username> <hostname> <port>
```

The command line parameters are used as follows:

`<username>`	the user 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 (sFtpS) is listening

All parameters are mandatory. An example is as follows:

sFtpC maclean eddie.cdf 1234

Your client should do the following:

- Make the connection to the specified server.
- Send an "authenticate" command
- Read the user's commands from the keyboard, does any required parsing, and sends the corresponding

commands to the server.

- The client exits when the user types "quit".

The Server

The synopsis for the server is as follows:

```
sFtpS
```

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 `sFtp`. 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 `sFtp` 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. Spawn a new thread to handle the connection. The new thread works as follows:

- The first command processed must be an authenticate command.
- After authentication, the server thread reads and processes commands until the client disconnects.

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 the RC4 encryption code to make this assignment simpler. All files will be found in `/u/csc209h/include` and `/u/csc209h/lib`.

`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. Note that I have changed this library since last term, so please use the up to date versions.

I will also provide source code for 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 (10 marks in total). 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.

I will also provide a utility to add new users to the file `authenticate.ftp`.

Some Requirements

- Use at least 10 `assert()` statements. These must be used properly, so read the style page on the course website if you are unsure.
- You must use `select()` in your client; marks will be deducted for improper use (if in doubt, ask).
- You may not use `fork()`/`exec()`; all 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 March 27th. That is, your client must work with my server, and your server must work with my client

Helpful Stuff

1. 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.

2. When you write data to a socket, don't write `NULL` characters that are not actually part of the data.

3. **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 an unencrypted client server that only supports authentication
   * add the "Directory Listing" command
   * add the "Change Directory" command
   * add the file transfer from server to client (this one is easier)
   * add the file transfer from client to server
   * add encryption [**note:** it may be best to encapsulate all of your reading/writing to the sockets in a single functions to make adding encryption easier at the end]

   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.

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

   Important: you must include these parameters before the name of the files to be linked.

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 `secureFtp.tar`. Use `gzip` to compress this file, thus creating `secureFtp.tar.gz`. It's probably a good idea to attempt uncompressing and un-`tar`-ing a copy of this file in a separate directory to make sure the `tar` operation worked.

You must hand in a printed version of your program (failure to submit a printout will result in a mark of 0/30 for

style), as well as submitting it electronically on CDF using "**`submit -N a4 csc209h secureFtp.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 50 marks for each of the client and server. For each part of the assignment 15 marks will be allocated for style, and 35 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. Write an unencrypted client/server that only supports authentication (30 marks).
2. Add the "Directory Listing" command to goal 1 (10 marks).
3. Add the "Change Directory" command to goal 2 (5 marks).
4. Add the file transfer from server to client to goal 3 (10 marks).
5. Add the file transfer from client to server to goal 4 (10 marks).
6. Add encryption. (5 marks)

The remaining 30 marks will be for style, so merely achieving goal 1 with excellent style will get you 60/100.