CS111 - Project 1B: I/O and IPC

INTRODUCTION:

In this project, you will build a multi-process telnet-like client and server. This project is a continuation of Project 1A. It can be broken up into two major steps:

- Passing input and output over a TCP socket
- Encrypting communication

RELATION TO READING AND LECTURES:

This lab will build on the process and exception discussions in lecture 3, but it is really about researching and exploiting APIs.

PROJECT OBJECTIVES:

- Demonstrate the ability to research new APIs and debug code that exploits them
- Do basic network communication
- Do basic encryption

DELIVERABLES:

A single tarball (.tar.gz) containing:

- two C source modules that compile cleanly (with no errors or warnings). You will have two source files, *client.c* and *server.c*, which should compile to the executables *client* and *server*, respectively.
- a Makefile to build the program and the tarball. The “make” command should build **both** *client* and *server*, and “make client” or “make server” should make the appropriate executable.
- a key file *my.key* containing the encryption key for step 2.

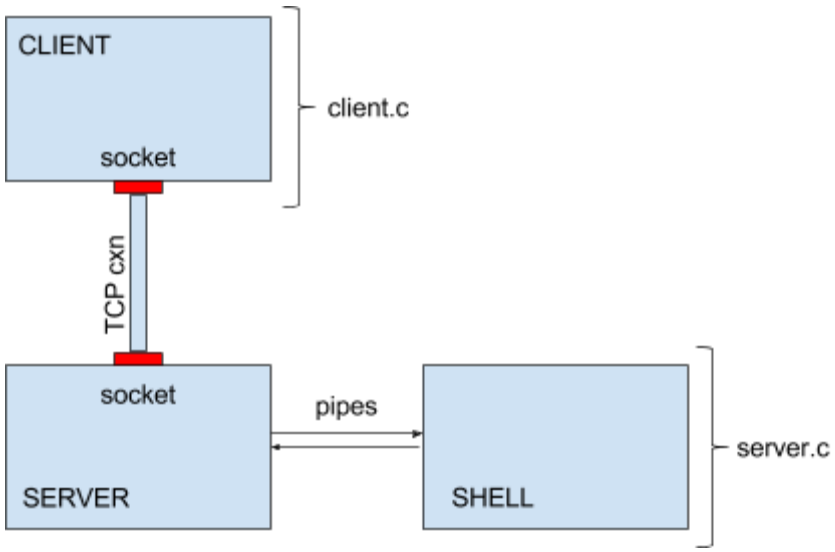
PROJECT DESCRIPTION:

Study the following man pages.

- *mcrypt(3)* ...a data encryption library (aka *libmcrypt*)
- *socket(7)* ...for creating communication endpoints (networking)

Passing input and output over a TCP socket.

Using Project 1A's step 2 (the --shell option) as a starting point, create a client program (*client.c*) and a server program (*server.c*), both of which support a --port=<portno> option. (Note there's no need to maintain the --shell option.)



Your overall program design will look like the diagram above. You will have a client process, a server process, and a shell process. The first two are connected via TCP connection, and the last two are connected via pipes.

- The client program will open a connection to the specified port (rather than sending it to a shell). The client should then send input from the keyboard to the socket (while echoing to the display), and input from the socket to the display.
    - Maintain the non-canonical terminal behavior from Project 1A.
    - Include a --log=<filename> option, available to the client only, which maintains a record of data sent over the socket (see below).
    - If a ^D is entered on the terminal, close the network connection, restore terminal modes, and exit the client with RC = 0. **ETA: ^C should behave like it does in the normal shell. I.e., do not unset the ISIG bit in your terminal modes.**
    - If the client gets a read error or EOF, close the network connection, restore terminal modes, and exit the client with RC = 1.
    - **ETA: include a --encrypt flag, which will turn on encryption (see below).**
- The server program will connect with the client, receive the client's commands and send them to the shell, and will "serve" the client the outputs of those commands. The server program will
    - listen on a network socket (port specified as a command line parameter)
    - accept a connection when it's made
    - once a connection is made, fork a child process which will exec a shell to process commands received; the server process should communicate with the shell via pipes

- redirect the shell process's stdin/stdout/stderr to the appropriate pipe ends (similar to what you did in Project 1A)
- redirect the server process's stdin/stdout/stderr to the socket
- read/write the shell input/output to the client via the socket. The shell should process commands received through the socket.
- If the server gets an EOF or SIGPIPE from the shell pipes (e.g., the shell exits), close the network connection, kill the shell, and exit with RC = 2.
- If the server gets a read error or EOF on the network connection, close the network connection, kill the shell, and exit with RC = 1.
- **ETA: include a --encrypt option, which will turn on encryption (see below).**

To send commands to the shell, you will need to run both the client and the server. On the client side, the behavior will be very similar to that of Project 1A, in that you will see your commands echoed back to the terminal as well as the output from the shell. Do not display anything on the server side.

This tutorial is a very detailed introduction to socket programming; it goes line-by-line through basic client and server code.

If you plan on developing on lnxsrv, please e-mail your TA to get a port number to avoid collisions. Otherwise, use a port number larger than 1023 (ports numbered lower than 1024 are reserved).

The --log=<filename> option.
On the client side, implement an option, --log=<filename>, which copies all data sent to and read from the client's socket and writes it to the specified file. Prefix all entries with "SENT X bytes" and "RECEIVED X bytes**:** " as appropriate. **ETA: remember to include the ": " (colon-space)!**

Sample log format output:

SENT 35 bytes: sendingsendingsendingsendingsending
RECEIVED 18 bytes: receivingreceiving

Encrypted communication.

- **ETA: include a --encrypt option in your client and server which, if included, will turn on encryption. Note on program execution you'll have to include it (or not) in both the server and the client.**
- Choose an encryption algorithm from Linux *libmcrypt*. Then modify both the client and server applications to encrypt traffic before sending it over the network and decrypt it after receiving it.
- Do encryption based on a key in a file called *my.key* in the current working directory, and include this key file with your submission.

- Using your key and encryption algorithm, run a session and use the --log option to verify that encryption and decryption are working properly. The --log option should record outgoing data post-encryption and incoming data pre-decryption.
- Do not include your --log files with your submission; we will generate our own from your program.

SUBMISSION:

Project 1B is due on Monday, April 18.
Your tarball should have a name of the form lab1b-*studentID*.tar.gz and should be submitted via CCLE.
We will test it on a SEASnet GNU/Linux server running RHEL 7. You would be well advised to test your submission on that platform before submitting it.

RUBRIC:

Value	Feature
	Packaging and Build (15%)
5%	untars expected contents
5%	clean build w/default action (no warnings)
3%	Makefile has working clean and dist targets
2%	reasonableness of README contents
	Communication (45%)
2%	Both programs accept the --port option
2%	**ETA: Both programs accept the --encrypt option**
14%	client program passes data between terminal and socket, reads shell output from server
14%	server program starts shell and reads/writes from/to shell (executes commands)
3%	^D at client closes network connection, restores terminal modes, and exits with RC = 0.
3%	EOF/error on either side of network closes connection, exits properly with RC = 1
2%	EOF or SIGPIPE at server **from shell pipes** exits with RC = 2.
5%	functioning --log=filename option
	Encryption (30%)
5%	Include key file & use key file for encryption-based communication
10%	Data is encrypted before sending
10%	Data is decrypted on receiving
5%	--log option records outgoing data post-encryption and incoming data pre-decryption
10%	**Code review (10%) -- checks that you're using the right libraries, etc.**