Operating Systems

Homework Assignment #4

Due 22.1.2017, 23:50

In this assignment, you will write a "cipher server" which acts similar to HW #1, over a socket, as well as a client which sends a file to the server and writes its output (the response from the server).

The server listens for connections. Whenever a client connects, the server forks a new process, which handles the client. Each process encrypts the data and writes the ciphertext back to the socket.

Submission:

- Make sure filenames are correct, case-sensitive, zip structure is correct.
- Submit an actual zip file created using the *zip* command **only!**
- **Submit 2 file:** `os_client.c`, **`os_server.c`**
- **Carefully follow all submission guidelines.**

Client

Command-line arguments:

1. IP – the IP address of the server (as a string, IPv4)
2. PORT – the port the server listens on (assume a positive short)
3. IN – the input file to read input from
4. OUT – the output file to write the output to

The flow:

The client performs the encryption (or decryption) operation by sending data from the input file IN to the server, and writing the result (the server's answer) into the output file OUT.

Once all data is sent, the client closes the connection, and then reads and handles any remaining data received from the server, and finishes.

- Data should be sent via a **TCP** socket, to the server specified by IP:PORT.
- You may assume that if X bytes are successfully sent to the server, then it replies with X bytes.
- You may not assume anything regarding the size of IN.
- IN must exist, otherwise output an error and exit.
- The path to the OUT file must exist, otherwise output an error and exit (i.e., no need to check the folder, just try to open/create the file).
- If OUT does not exist, the client should create it. If it exists, it should truncate it.

Server

Command-line arguments:

1. PORT – the port to listen for connections on (assume a positive short)
2. KEY – path to a key file
3. KEYLEN – length of the key file that should be generated (assume a positive integer, optional)

The flow:

1. If KEYLEN is provided, initialize the key file:
    a. Create a new file or truncate an existing file – KEY
    b. Use the `/dev/urandom` file to write KEYLEN random bytes into KEY
2. Create a TCP socket that listens on PORT (use 10 as the parameter for *listen*).
3. Register a signal handler for SIGINT:
    a. If SIGINT is received, the server should cleanup and exit gracefully.
4. Wait for connections (i.e., *accept* in a loop).
5. Whenever a client connects, fork a new process.
6. The parent process should continue accepting connections.
7. In the new process:
    a. Open the key file.
    b. Read data from the client until EOF.
    c. Whenever data is read, encrypt, and send back to the client.
    d. When finished, close the socket, cleanup, and exit gracefully.

Notes:

- Each forked process opens the key file separately. The main (parent) process does not open the key file at all, except when initializing it (if KEYLEN is provided).
- You cannot assume that the key file exists or has data. In either case, if KEYLEN is not provided, the server should fail starting (exit with an error).
- You cannot assume anything regarding the overall size of the key file or the input.
- You should try to be as efficient as possible in receiving, encrypting, and sending data; Namely, when any data block is received, then encrypt it immediately and send it back, don't wait for *more data* from the client, i.e., don't aggregate encryption requests.
- Errors in a child process should terminate the child process *only*.