

Program 4 – CS 344

Overview

In this assignment, you will be creating five small programs that encrypt and decrypt information using a one-time pad-like system. I believe that you will find the topic quite fascinating: one of your challenges will be to pull yourself away from the stories of real-world espionage and tradecraft that have used the techniques you will be implementing.

These programs serve as a capstone to what you have been learning in this course, and will combine the multi-processing code you have been learning with socket-based inter-process communication. Your programs will also be accessible from the command line using standard UNIX features like input/output redirection, and job control. Finally, you will write a short compilation script.

Specifications

All execution, compiling, and testing of this program should ONLY be done in the bash prompt on our class server!

Use the following link as your primary reference on One-Time Pads (OTP):

http://en.wikipedia.org/wiki/One-time_pad (Links to an external site.)

The following definitions will be important:

Plaintext is the term for the information that you wish to encrypt and protect. It is human readable.

Ciphertext is the term for the plaintext after it has been encrypted by your programs. Ciphertext is not human-readable, and in fact cannot be cracked, if the OTP system is used correctly.

A **Key** is the random sequence of characters that will be used to convert Plaintext to Ciphertext, and back again. It must not be re-used, or else the encryption is in danger of being broken.

Your program will encrypt and decrypt plaintext into ciphertext, using a key, in exactly the same fashion as above, except it will be using modulo 27 operations: your 27 characters are the 26 capital letters, and the space character (). All 27 characters will be encrypted and decrypted as above.

To do this, you will be creating five small programs in C. Two of these will function like "daemons" (but aren't actually daemons), and will be accessed using network sockets. Two will use the daemons to perform work, and the last is a standalone utility.

Your programs must use the network calls we've discussed in class (`send()`, `recv()`, `socket()`, `bind()`, `listen()`, & `accept()`) to send and receive sequences of bytes for the purposes of encryption and decryption by the appropriate daemons. The whole point is to use the network, even though for testing purposes we're using the same machine: if you just `open()` the datafiles from the server without using the network calls, you'll receive 0 points on the assignment.

Here are the specifications of the five programs:

otp_enc_d: This program will run in the background as a daemon. Upon execution, `otp_enc_d` must output an error if it cannot be run due to a network error, such as the ports being unavailable. Its function is to perform the actual encoding, as described above in the Wikipedia quote. This program will listen on a particular port/socket, assigned when it is first ran (see syntax below). When a connection is made, `otp_enc_d` must call `accept()` to generate the socket used for actual communication, and then use a separate process to handle the rest of the transaction (see below), which will occur on the newly accepted socket.

This child process of `otp_enc_d` must first check to make sure it is communicating with `otp_enc` (see `otp_enc`, below). After verifying that the connection to `otp_enc_d` is coming from `otp_enc`, then this child receives from `otp_enc` plaintext and a key via the communication socket (not the original listen socket). The `otp_enc_d` child will then write back the ciphertext to the `otp_enc` process that it is connected to via the same communication socket. Note that the key passed in must be at least as big as the plaintext.

Your version of `otp_enc_d` must support up to five *concurrent* socket connections running at the same time; this is different than the number of processes that could queue up on your listening socket (which is specified in the second parameter of the `listen()` call). Again, only in the child process will the actual encryption take place, and the ciphertext be written back: the original server daemon process continues listening for new connections, not encrypting data.

In terms of creating that child process as described above, you may either create with `fork()` a *new* process every time a connection is made, *or* set up a pool of five processes at the beginning of the program, before connections are allowed, to handle your encryption tasks. As above, your system must be able to do five separate encryptions at once, using either method you choose.

Use this syntax for `otp_enc_d`:

```
otp_enc_d listening_port
```

listening_port is the port that `otp_enc_d` should listen on. You will always start `otp_enc_d` in the background, as follows (the port 57171 is just an example; yours should be able to use any port):

```
$ otp_enc_d 57171 &
```

In all error situations, this program must output errors to *stderr* as appropriate (see grading script below for details), but should not crash or otherwise exit, unless the errors happen when the program is starting up (i.e. are part of the networking start up protocols like `bind()`). If given bad input, once running, `otp_enc_d` should recognize the bad input, report an error to *stderr*, and continue to run. Generally speaking, though, this daemon shouldn't receive bad input, since that should be discovered and handled in the client first. All error text must be output to *stderr*.

This program, and the other 3 network programs, should use "localhost" as the target IP address/host. This makes them use the actual computer they all share as the target for the networking connections.

otp_enc: This program connects to `otp_enc_d`, and asks it to perform a one-time pad style encryption as detailed above. By itself, `otp_enc` doesn't do the encryption - `otp_enc_d` does. The syntax of `otp_enc` is as follows:

If `otp_enc` receives key or plaintext files with ANY bad characters in them, or the key file is shorter than the plaintext, then it should terminate, send appropriate error text to *stderr*, and set the exit value to 1.

`otp_enc` should NOT be able to connect to `otp_dec_d`, even if it tries to connect on the correct port - you'll need to have the programs reject each other. If this happens, `otp_enc` should report the rejection to *stderr* and then terminate itself. In more detail: if `otp_enc` cannot connect to the `otp_enc_d` server, for any reason (including that it has accidentally tried to connect to the `otp_dec_d` server), it should report this error to *stderr* with the attempted port, and set the exit value to 2. Otherwise, upon successfully running and terminating, `otp_enc` should set the exit value to 0.

Again, any and all error text must be output to *stderr* (not into the plaintext or ciphertext files).

otp_dec_d: This program performs exactly like `otp_enc_d`, in syntax and usage. In this case, however, `otp_dec_d` will decrypt ciphertext it is given, using the passed-in ciphertext and key. Thus, it returns plaintext again to `otp_dec`.

otp_dec: Similarly, this program will connect to `otp_dec_d` and will ask it to decrypt ciphertext using a passed-in ciphertext and key, and otherwise performs exactly like `otp_enc`, and must be runnable in the same three ways. `otp_dec` should NOT be able to connect to `otp_enc_d`, even if it tries to connect on the correct port - you'll need to have the programs reject each other, as described in `otp_enc`.

keygen: This program creates a key file of specified length. The characters in the file generated will be any of the 27 allowed characters, generated using the standard UNIX randomization methods. Do not create spaces every five characters, as has been historically done. Note that you specifically do not have to do any fancy random number generation: we're not looking for cryptographically secure random number generation! `rand()` is just fine. The last character `keygen` outputs should be a newline. All error text must be output to *stderr*, if any.

Files and Scripts

You are provided with 5 plaintext files to use ([one](#), [two](#), [three](#), [four](#), [five](#)). The grading will use these specific files; do not feel like you have to create others.

Example

Here is an example of usage, if you were testing your code from the command line:

```
$ cat plaintext1
THE RED GOOSE FLIES AT MIDNIGHT STOP
$ otp_enc_d 57171 &
$ otp_dec_d 57172 &
$ keygen 10
EONHQCKQ I
$ keygen 10 > mykey
$ cat mykey
VAONWOYVXP
$ keygen 10 > myshortkey
$ otp_enc plaintext1 myshortkey 57171 > ciphertext1
Error: key 'myshortkey' is too short
$ echo $?
1
$ keygen 1024 > mykey
$ otp_enc plaintext1 mykey 57171 > ciphertext1
$ cat ciphertext1
WANAWTRLFTTH RAAQGZSOHCTYS JDBEGYZQDQ
$ keygen 1024 > mykey2
$ otp_dec ciphertext1 mykey 57172 > plaintext1_a
$ otp_dec ciphertext1 mykey2 57172 > plaintext1_b
$ cat plaintext1_a
THE RED GOOSE FLIES AT MIDNIGHT STOP
$ cat plaintext1_b
WSXFHCJAEISWQRNO L ZAGDIAUAL IGGTKBW
$ cmp plaintext1 plaintext1_a
$ echo $?
0
$ cmp plaintext1 plaintext1_b
plaintext1 plaintext1_b differ: byte 1, line 1
$ echo $?
1
$ otp_enc plaintext5 mykey 57171
otp_enc error: input contains bad characters
$ echo $?
1
$ otp_enc plaintext3 mykey 57172
Error: could not contact otp_enc_d on port 57172
$ echo $?
2
$
```