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 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 the

eos-class.engr.oregonstate.edu 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.

The following excerpt from this Wikipedia article was captured on 2/21/2015:

“Suppose Alice wishes to send the message "HELLO" to Bob. Assume two pads of paper containing identical random sequences of letters were somehow previously produced and securely issued to both. Alice chooses the appropriate unused page from the pad. The way to do this is normally arranged for in advance, as for instance 'use the 12th sheet on 1 May', or 'use the next available sheet for the next message'.

The material on the selected sheet is the key for this message. Each letter from the pad will be combined in a predetermined way with one letter of the message. (It is common, but not required, to assign each letter a numerical value, e.g., "A" is 0, "B" is 1, and so on.)

In this example, the technique is to combine the key and the message using modular addition. The numerical values of corresponding message and key letters are *added* together, modulo 26. So, if key material begins with "XMCKL" and the message is "HELLO", then the coding would be done as follows:

```
      H     E     L     L     O  message
   7 (H)   4 (E)  11 (L)  11 (L)  14 (O) message
+ 23 (X)  12 (M)   2 (C)  10 (K)  11 (L) key
= 30  16    13    21    25    message + key
=  4 (E)  16 (Q)  13 (N)  21 (V)  25 (Z) message + key (mod 26)
      E     Q     N     V     Z  → ciphertext
```

If a number is larger than 26, then the remainder, after *subtraction* of 26, is taken [as the result]. This simply means that if the computations "go past" Z, the sequence starts again at A.

The ciphertext to be sent to Bob is thus "EQNVZ". Bob uses the matching key page and the same process, but in reverse, to obtain the plaintext. Here the key is *subtracted* from the ciphertext, again using modular arithmetic:

```
      E     Q     N     V     Z  ciphertext
      4 (E)  16 (Q)  13 (N)  21 (V)  25 (Z) ciphertext
-  23 (X)  12 (M)   2 (C)  10 (K)  11 (L) key
= -19       4     11    11    14    ciphertext - key
=   7 (H)   4 (E)  11 (L)  11 (L)  14 (O) ciphertext - key (mod 26)
      H     E     L     L     O  → message
```

Similar to the above, if a number is negative then 26 is *added* to make the number zero or higher.

Thus Bob recovers Alice's plaintext, the message "HELLO". Both Alice and Bob destroy the key sheet immediately after use, thus preventing reuse and an attack against the cipher.”

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.

Here are the specifications of the five programs:

`otp_enc_d:` This program will run in the background as a daemon. Its function is to perform the actual encoding, as descripted above in the Wikipedia quote. This program will listen on a particular port, assigned when it is first ran, and receives plaintext and a key via that port when a connection to it is made. It will then write back the ciphertext to the process that it is connected to via the same port. Note that the key passed in must be at least as big as the plaintext. This program must output an error if the program cannot be run due to a network error, such as the ports being unavailable.

When `otp_enc_d` makes a connection with `otp_enc`, it must fork off a separate process immediately, and be available to receive more connections. Your version must support up to five concurrent socket connections. In the forked off process, the actual encryption will take place, and the ciphertext will be written back.

You may either create a new process every time a connection is made, or set up a pool of five process to handle your encryption tasks. 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, your program must output errors 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. That is, if given bad input, once running, `otp_enc_d` should recognize the bad input, report an error to the screen, and continue to run.

Your `otp_enc_d` should be killable with the `-KILL` signal, as normal: you may not have to do anything for your program to have this ability.

This program, and the other 3 network programs, should use "localhost" as the target IP address/host. This makes them use eos-class directly as the host they're all running on.

`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. Its syntax is as follows:

`otp_enc` *`plaintext key port`*

In this syntax, *`plaintext`* is the name of a file in the current directory that contains the plaintext you wish to encrypt. Similarly, *`key`*contains the encryption key you wish to use to encrypt the text. Finally, *`port`* is the port that `otp_enc` should attempt to connect to`otp_enc_d` on.

When `otp_enc` receives the ciphertext, it should output it to *stdout*. Thus, `otp_enc` can be launched in any of the following methods, and should send its output appropriately:

```
$ otp_enc myplaintext mykey 57171

$ otp_enc myplaintext mykey 57171 > myciphertext

$ otp_enc myplaintext mykey 57171 > myciphertext &
```

If `otp_enc` receives key or plaintext files with bad characters in them, or the key file is shorter than the plaintext, it should exit with an error, and set the exit value to 1. This character validation can happen in either `otp_enc` or `otp_enc_d`, your choice. If `otp_enc`cannot find the port given, it should report this error to the screen (not into the plaintext or ciphertext files) with the bad port, and set the exit value to 2. Otherwise, on successfully running, `otp_enc` should set the exit value to 0. `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.

`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`. This program must output an error if the program cannot be run due to a network error, such as the ports being unavailable.

`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. It will use the same syntax and usage as `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.

`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!

The syntax for keygen is as follows:

`keygen` *`keylength`*

Where *`keylength`* is the length of the key file in characters. `keygen` outputs to stdout. Here is an example run, which redirects stdout to a key file of 256 characters called "mykey":

```
$ keygen 256 > mykey
```