Program 4 - OTP

Submit Assignment

Due Aug 11 by 2:59am   **Points** 160   **Submitting** a file upload
Available Jul 27 at 3:01am - Aug 13 at 2:59am 17 days

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   (http://en.wikipedia.org/wiki/One-time_pad)

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 (note that the C % operator won't perform the modulus you want - you'll have to write your own). 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, this 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 (i.e. are part of the networking protocols). 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. All error text must be output to stderr.

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.

All error text must be output to stderr.

`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 last character keygen outputs should be a newline. All error text must be output to stderr, if any.

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" (note that mykey is 257 characters long because of the newline):

```
$ keygen 256 > mykey
```

Files and Scripts

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

You are also provided with a grading script ("<u>**p4gradingscript**</u> ☒") that you can run to test your software. If it passes the tests in the script, and has sufficient commenting, it will receive full points (see below). EVERY TIME you run this script, change the port numbers you use! Otherwise, because UNIX may not let go of your ports immediately, your successive runs may fail!

Finally, you will be required to write a compilation script (see below) that compiles all five of your programs, allowing you to use whatever C code and methods you desire. This will ease grading. Note that only C will be allowed, no C++ or any other language (Python, Perl, awk, etc.).

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
```

```
$ otp_enc_d 57171 &
$ otp_dec_d 57172 &
$ 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
GU WIRGEWOMGRIFOENBYIWUG T WOFL
$ 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
$ cat plaintext1_b
WVIOWBTUEIOBC  FVTROIROUXA JBWE
$ 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_d error: input contains bad characters
$ otp_enc plaintext3 mykey 57172
Error: could not contact otp_enc_d on port 57172
$ echo $?
2
$
```

Compilation Script

You must also write a short bash shell script called "compileall" that merely compiles your five programs. For example, the first two lines might be:

```
#!/bin/bash
gcc -o otp_enc_d otp_enc_d.c
…
```

This script will be used to compile your software, and must successfully run on our class server. The compilation must create all five programs, in the same directory as "compileall", for immediate use by the grading script, which is named "p4gradingscript".

What to Submit

Please submit a single zip file of your program code, which may be in as many different files as you want. Inside that zip file, include the following files

1. All of your program code

2. The compilation script named "compileall"

3. All five plaintext# files, numbered 1 through 5

4. A copy of the grading script named "p4gradingscript"

Failing to submit one of the required pieces results in an 8-point deduction, while we attempt to contact you to submit what's missing. Your submission date & time is whenever you send in the mission piece.

Hints

Staring Points

Start with the sample network programs **client.c** 📄 🔗 and **server.c** 📄 🔗 as described at **linuxhowtos.org (http://www.linuxhowtos.org/C_C++/socket.htm)** . These compile and function! Base your code off of these.

If you have questions about what your programs needs to be able to do, just examine the grading script. Your programs have to deal with exactly what's in there: no more, no less. :)

Sending Data

Recall that when sending data, not all of the data may be written with just one call to send() or write(). This occurs because of network interruptions, server load, and other factors. You'll need to carefully watch the number of characters read and/or written, as appropriate. If the number returned is less than what you intended, you'll need to restart the process from where it stopped. This means you'll need to wrap a loop around the send/receive routines to ensure they finish their job before continuing.

If you try to send too much data at once, the server will likely break the transmission, as in the previous paragraph. Consider setting a maximum send size, breaking the transmission yourself every 1000 characters, say.

There are a few ways to handle knowing how much data you need to send in a given transmission. One way is to send an integer from client to server (or vice versa) first, informing the other side how much is coming. This relatively small integer is unlikely to be split and interrupted. Another way is to have the listening side looking for a termination character that it recognizes as the end of the transmission string. It could loop, for example, until it has seen that termination character.

Concurrency Implications

Remember that only one socket can be bound to a port at a time. Multiple incoming connections all queue up on the socket that has had listen() called on it for that port. After each accept() call is made, a new socket file descriptor is returned which is your server's handle to that TCP connection. The server can accept multiple incoming streams, and communicate with all of them, by continuing to call accept(), generating a new socket file descriptor each time - except, in our implementation, we're going to add an additional wrinkle.

In our program, we're going to pretend/enforce that only one TCP connection can be live on each port at a time. Thus, you'll have to have the server handing off the processing jobs to other newly forked child processes of itself on different ports, leaving the main port open.

The incoming connections all queue up on the primary listening port, and you can loop accept() on the socket in the server to process each one in turn. When dealing with a new connection from a client, have the server and client first agree what new port they should be talking on. Then, have the client close the connection and restart it on the newly agreed upon port. Likewise, the server forks a copy of itself, which uses the new port and waits for the incoming connection from the handled client! In this way, the listening port indicated on the command line when the server was started is always open and available to handle new connections rapidly.

About Newlines

You are only supposed to accept the 26 letters of alphabet and the "space" character as valid for encrypting/decrypting. However, all of the plaintext input files end with a newline character. *Text files need to end in a newline character for various reasons.*

When one of your programs reads in an input file, strip off the newline. Then encrypt and decrypt the text string, again with no newline character. *When you send the result to stdout, or save results into a file, tack a newline to the end, or your length will be off in the grading script.* Note that the newline character affects the length of files as reported by the wc command! Try it!

About Reusing Sockets

In the p4gradingscript, you can select which ports to use: I recommend ports in the 50000+ range. However, UNIX doesn't immediately let go of the ports you use after your program finishes! I highly recommend that you frequently change and randomize the sockets you're using, to make sure you're not using sockets that someone else is playing with. In addition, to allow your program to continue to use the same port (your mileage may vary), read this:

http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html#setsockoptman (http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html#setsockoptman)

...and then play around with this command:

```
setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));
```

Where to Develop

Finally, I HIGHLY recommend that you develop this program directly on the eos-class server. Doing so will prevent you from having problems transferring the program back and forth, which can cause compatibility issues.

If you do see ^M characters all over your files, try this command:

```
$ dos2unix bustedFile
```

Grading

The graders will run the "compileall" script, and will then run the "p4gradingscript". They will make a reasonable effort to make your code compile, but if it doesn't compile, you'll receive a zero on this assignment. If it compiles, it will have the "p4gradingscript" script ran against it for final grading, in this manner, in a bash prompt on eos-class:

```
$ ./p4gradingscript PORT1 PORT2 > mytestresults 2>&1
```

The graders will change the ports around each time they run the grading script, to make sure the ports used aren't in-use. Points will be assigned according to this grading script.

150 points are available in the grading script, while the final 10 points will be based on your style, readability, and commenting. Comment well, often, and verbosely: we want to see that you are telling us WHY you are doing things, in addition to telling us WHAT you are doing.

The TAs will use this exact set of instructions: **Program4 Grading.pdf** to grade your submission.