Assignment 4: One-Time Pads

Due Mar 6 by 11:59pm **Points** 160 **Submitting** a file upload **File Types** zip
Available Jan 9 at 8am - Mar 22 at 11:59pm

This assignment was locked Mar 22 at 11:59pm.

Introduction

In this assignment, you will be creating five small programs that encrypt and decrypt information using a one-time pad-like system. These programs 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.

Learning Outcomes

After successful completion of this assignment, you should be able to do the following

- Compare and contrast IPC facilities for communication (Module 7, MLO 2)
- Explain the Client-Server communication model at a high level (Module 8, MLO 1)
- Understand and use the programmer's view of the internet to design network programs (Module 8, MLO 3)
- Explain the concept of Unix sockets (Module 8, MLO 4)
- Design and implement client and server programs for IPC using sockets (Module 8, MLO 5)
- Compare and evaluate designs for servers (Module 8, MLO 6)

One-Time Pads

Use the wikipedia page **One-Time Pads (http://en.wikipedia.org/wiki/One-time_pad)** as your primary reference on One-Time Pads (OTP).

Definitions

- **Plaintext**: The information that you wish to encrypt and protect. It is human readable.
- **Ciphertext**: Plaintext *after* it has been encrypted by your programs. Ciphertext is not human-readable, and if the OTP system is used correctly, cannot be cracked.
- **Key**: A 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.

Example

The following example is from the above Wikipedia article.

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 25, then the remainder after subtraction of 26 is taken in modular arithmetic fashion. 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.

Specifications

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 as servers, and will be accessed using network sockets. Two will be clients, each one of these will use one of the servers to perform work, and the last program is a standalone utility.

Your programs must use the API for network IPC that we have discussed in the class (`socket`, `connect`, `bind`, `listen`, & `accept` to establish connections; `send`, `recv` to send and receive sequences of bytes) for the purposes of encryption and decryption by the appropriate servers. The whole point is to use the network, even though for testing purposes we're using the same machine to run all the programs: 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:

enc_server

This program is the encryption server and will run in the background as a daemon.

- 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).
- Upon execution, `enc_server` must output an error if it cannot be run due to a network error, such as the ports being unavailable.
- When a connection is made, `enc_server` must call `accept` to generate the socket used for actual communication, and then use a separate process to handle the rest of the servicing for this client connection (see below), which will occur on the newly accepted socket.
- This child process of `enc_server` must first check to make sure it is communicating with `enc_client` (see `enc_client`, below).
- After verifying that the connection to `enc_server` is coming from `enc_client`, then this child receives plaintext and a key from `enc_client` via the connected socket.
- The `enc_server` child will then write back the ciphertext to the `enc_client` process that it is connected to via the same connected socket.
- Note that the key passed in must be at least as big as the plaintext.

Your version of `enc_server` must support up to five concurrent socket connections running at the same time; this is different than the number of client connection requests that could queue up on your listening socket (which is specified in the second parameter of the `listen` call). Again, only in the child server 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 a new process with `fork` when a connection is made, or set up a pool of five processes at the beginning of the program before the server allows connections. Regardless of the method you choose, your system must be able to do five separate encryptions at once.

Use this syntax for `enc_server`:

```
enc_server listening_port
```

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

```
$ enc_server 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`). Once running, `enc_server` should recognize any bad input it receives, report an error to `stderr`, and continue to run. Generally speaking, though, this server 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.

enc_client

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

```
enc_client 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 `enc_client` should attempt to connect to `enc_server` on. When `enc_client` receives the ciphertext back from `enc_server`, it should output it to `stdout`. Thus, `enc_client` can be launched in any of the following methods, and should send its output appropriately:

```
$ enc_client myplaintext mykey 57171
$ enc_client myplaintext mykey 57171 > myciphertext
$ enc_client myplaintext mykey 57171 > myciphertext &
```

If `enc_client` 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.

`enc_client` should NOT be able to connect to `dec_server`, even if it tries to connect on the correct port - you'll need to have the programs reject each other. If this happens, `enc_client` should report the rejection to `stderr` and then terminate itself. In more detail: if `enc_client` cannot connect to the `enc_server` server, for any reason (including that it has accidentally tried to connect to the `dec_server` 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, `enc_client` 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).

dec_server

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

dec_client

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

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() (https://man7.org/linux/man-pages/man3/rand.3.html)** is just fine. The last character keygen outputs should be a newline. Any error text must be output to stderr.

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 creates a key of 256 characters and redirects `stdout` a file 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 (one, two, three, four, five). The grading will use these specific files; do not feel like you have to create others.

- **plaintext1 (https://canvas.oregonstate.edu/courses/1901760/files/94284879/download?wrap=1) (https://canvas.oregonstate.edu/courses/1901760/files/94284879/download?download_frd=1)**
- **plaintext2 (https://canvas.oregonstate.edu/courses/1901760/files/94284880/download?wrap=1) (https://canvas.oregonstate.edu/courses/1901760/files/94284880/download?download_frd=1)**

- **plaintext3 (https://canvas.oregonstate.edu/courses/1901760/files/94284865/download?wrap=1) (https://canvas.oregonstate.edu/courses/1901760/files/94284865/download?download_frd=1)**
- **plaintext4 (https://canvas.oregonstate.edu/courses/1901760/files/94284881/download?wrap=1) (https://canvas.oregonstate.edu/courses/1901760/files/94284881/download?download_frd=1)**
- **plaintext5 (https://canvas.oregonstate.edu/courses/1901760/files/94284874/download?wrap=1) (https://canvas.oregonstate.edu/courses/1901760/files/94284874/download?download_frd=1)**

You are also provided with a grading script `p5testscript` (https://canvas.oregonstate.edu/courses/1901760/files/94284875/download) that you can run to test your software. If it passes the tests in the script, and your code has sufficient commenting, your assignment will receive full points. The file **assignment5-otp-list-of-tests.pdf (https://canvas.oregonstate.edu/courses/1901760/files/94284910/download?wrap=1)** provides you with a list of tests included in the test script and the points corresponding to these tests.

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 (or use the one provided by us, see below) that compiles all five of your programs.

Example Usage

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
$ enc_server 57171 &
$ dec_server 57172 &
$ keygen 10
EONHQCKQ I
$ keygen 10 > mykey
$ cat mykey
VAONWOYVXP
$ keygen 10 > myshortkey
$ enc_client plaintext1 myshortkey 57171 > ciphertext1
Error: key ‘myshortkey’ is too short
$ echo $?
1
$ keygen 1024 > mykey
$ enc_client plaintext1 mykey 57171 > ciphertext1
$ cat ciphertext1
WANAWTRLFTH RAAQGZSOHCTYS JDBEGYZQDQ
$ keygen 1024 > mykey2
$ dec_client ciphertext1 mykey 57172 > plaintext1_a
$ dec_client 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
$ enc_client plaintext5 mykey 57171
enc_client error: input contains bad characters
$ echo $?
1
$ enc_client plaintext3 mykey 57172
Error: could not contact enc_server on port 57172
$ echo $?
2
$
```

Compilation Script

You can have as many C files for your programs as you want. You must also submit a bash shell script called `compileall` that creates 5 executable programs from your files. These 5 programs must be created in the same directory as `compileall`. The programs must be named `enc_server`, `enc_client`, `dec_server`, `dec_client` and `keygen`.

If you have only 5 C files, each with the same name as the executable program it will produce, you can use and submit the following shell script as your `compileall` script:

```
#!/bin/bash
gcc -o enc_server enc_server.c
gcc -o enc_client enc_client.c
gcc -o dec_server dec_server.c
gcc -o dec_client dec_client.c
gcc -o keygen keygen.c
```

Note: You are allowed to submit a `Makefile` instead of a `compileall` script. However, running the `Makefile` must create the 5 executable files with the names specified above and these files must be created in the same directory as your `Makefile`.

Hints

Where to Start

First, write `keygen` - it's simple and fun! Then, use our sample network programs **client.c (https://repl.it/@cs344/83clientc?lite=true#client.c)** and **server.c (https://repl.it/@cs344/83serverc?lite=true#server.c)** (you don't have to cite your use of them) to implement `enc_client` and `enc_server`. Once they are functional, copy them and begin work on `dec_client` and `dec_server`.

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 and Receiving Data

Recall that when sending data, not all of the data may get written with just one call to `send`. Similarly, when receiving data, not all the data may be read by one call to `recv`. 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. 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.

About Newlines

You are only supposed to accept the 26 letters of alphabet and the "space" character as valid for encrypting and decrypting. However, all of the plaintext input files end with a newline character, and all text files you generate must end in a newline character.

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, you must 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 file `p5testscript`, 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 ports you're using, to make sure you're not using ports that someone else is playing with. In addition, to allow your program to continue to use the same port (your mileage may vary), read the man page for `setsockopt` at **[Beej's Guide to Network Programming](https://beej.us/guide/bgnet/html/#setsockoptman) (https://beej.us/guide/bgnet/html/#setsockoptman)** and then play around with this function:

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

What to turn in?

- You can only use C for coding this assignment and you must use the gcc compiler.
- You can use C99 or GNU99 standard or the default standard used by the gcc installation on os1.
- Your assignment will be graded on os1.
- Submit a single zip file containing the following.
  1. All of your program code, which can be in as many different files as you want
  2. The compilation script named `compileall` or a `Makefile`. Even if you are using the `compileall` provided by us, you must include it in your submission.
  3. All five plaintext# files, numbered 1 through 5.
  4. A copy of the grading script named `p5testscript`.
- This zip file must be named `youronid_program4.zip` where youronid must be replaced by your own ONID.
  - E.g., if chaudhrn was submitting the assignment, the file must be named `chaudhrn_program4.zip`.
- When you resubmit a file in Canvas, Canvas can attach a suffix to the file, e.g., the file name may become `chaudhrn_program4-1.zip`. Don't worry about this name change as no points will be deducted because of this.

Grading

In a bash prompt, on our class server, the graders will run the `compileall` script (or your `Makefile`), and will then run the `p5testscript`. 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, then `p5testscript` script will be run for final grading in a bash prompt on our class server os1 in the following manner (where numbers are filled in for RANDOM_PORT1 and RANDOM_PORT2)

```
$ ./p5testscript RANDOM_PORT1 RANDOM_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 (at least every five lines, say): we want to see that you are telling us WHY you are doing things, in addition to telling us WHAT you are doing.

Assignment 5 Rubric

Criteria	Ratings		Pts
Points from the test script	**150 pts** **Full Marks**	**0 pts** **No Marks**	150 pts
The code is fully commented	**10 pts** **Full Marks**	**0 pts** **No Marks**	10 pts
			Total Points: 160