Available Nov 18 at 12:01am - Dec 4 at 11:59pm 17 days

Program 4 - OTP Due Dec 2 by 11:59pm

**Program 4 – CS 344** 

回

₹ Groups

**□**)|←

Files

Quizzes

Modules

Assignments Announcements

Fall 2016

Syllabus

Discussions Pages

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

Submitting a file upload

Submit Assignment

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

Media Gallery http://en.wikipedia.org/wiki/One-time\_pad 2\*

Specifications

The following definitions will be important:

My Media

Threadz

coding would be done as follows:

Here are the specifications of the five programs:

57171 is just an example; yours should be able to use any port);

\$ otp\_enc myplaintext mykey 57171 > myciphertext &

cannot be cracked, if the OTP system is used correctly.

"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

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

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

0 message 7 (H) 4 (E) 11 (L) 11 (L) 14 (O) message

```
= 4 (E) 16 (Q) 13 (N) 21 (V) 25 (Z) message + key (mod 26)
                                               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
```

23 (X) 12 (M) 2 (C) 10 (K) 11 (L) key

be accessed using network sockets. Two will use the daemons to perform work, and the last is a standalone utility.

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 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

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

In the child process of otp\_enc\_d, it 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. Again, only in the child process will the actual

encryption take place, and the ciphertext be written back. 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

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

otherwise exit, unless the errors happen when the program is starting up (i.e. are part of the networking start up protocols like bind()). 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

In all error situations, this program must output errors as appropriate (see grading script below for details), but should not crash or

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,  $\verb"otp_enc" doesn't do the encryption - \verb"otp_end_d" does. The syntax of \verb"otp_enc" is as follows:$ otp\_enc plaintext key port

\$ otp\_enc\_d 57171 &

When otp\_enc receives the ciphertext back from otp\_enc\_d, 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

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. If this happens, otp\_enc should report the rejection and then terminate itself.

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

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

text must be output to stderr, if any. The syntax for keygen is as follows:

Files and Scripts You are provided with 5 plaintext files to use (one of, two of, three of, four of, five of). The grading will use these specific files; do not feel

keygen keylength

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,

You are also provided with a grading script ("p4gradingscript &") that you can run to test your software. If it passes the tests in the script,

THE RED GOOSE FLIES AT MIDNIGHT \$ otp\_enc\_d 57171 & \$ otp\_dec\_d 57172 &

Error: key 'myshortkey' is too short \$ echo \$? \$ keygen 1024 > mykey

\$ keygen 10 > myshortkey

etc.).

\$ otp\_enc plaintext1 mykey 57171 > ciphertext1 \$ cat ciphertext1 GU WIRGEWOMGRIFOENBYIWUG T WOFL \$ keygen 1024 > mykey2

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"

Hints Staring Points

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

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.

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.

Your submission date & time is whenever you send in the missing piece.

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

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

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.

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:

**About Reusing Sockets** 

setsockopt(sock\_fd, SOL\_SOCKET, SO\_REUSEADDR, &yes, sizeof(int)); Where to Develop

\$ dosZunix 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

If you do see ^M characters all over your files, which come from copying a Windows-saved file onto a UNIX file system, try this command:

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 our class server

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

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.

Overview

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

UNIX features like input/output redirection, and job control. Finally, you will write a short compilation script.

Points 160

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

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

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:

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.)

+ 23 (X) 12 (M) 2 (C) 10 (K) 11 (L) key = 30 16 13 21 25 mess message + key

plaintext. Here the key is subtracted from the ciphertext, again using modular arithmetic: Z ciphertext 4 (E) 16 (Q) 13 (N) 21 (V) 25 (Z) ciphertext = -19 11 11 14 ciphertext - key = 7 (H) 4 (E) 11 (L) 11 (L) 14 (O) ciphertext - key (mod 26) 0 → message

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 (note that the C operator "%" won't perform the modulus you want - you'll have to write your own). This program will (see below), which will occur on the newly accepted socket.

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

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

In this syntax, plaintext is the name of a file in the current directory that contains the plaintext you wish to encrypt. Similarly, key  $contains the encryption key you wish to use to encrypt the text. Finally, \textit{post} is the port that \texttt{otp\_enc} should attempt to connect to the port that \texttt{otp\_enc} should attempt to connect to the port that \texttt{otp\_enc} should attempt to connect to the port that \texttt{otp\_enc} should attempt to connect to the port that \texttt{otp\_enc} should attempt to connect to the port that \texttt{otp\_enc} should attempt to connect to the port that \texttt{otp\_enc} should attempt to connect to the port that \texttt{otp\_enc} should attempt to connect to the port that \texttt{otp\_enc} should attempt to connect to the port that \texttt{otp\_enc} should attempt to connect to the port that \texttt{otp\_enc} should attempt to connect to the port that \texttt{otp\_enc} should attempt to connect to the port that \texttt{otp\_enc} should attempt to connect to the port that \texttt{otp\_enc} should attempt to the port t$ 

All error text must be output to stderr.

otp\_enc.

cryptographically secure random number generation! rand () is just fine. The last character keygen outputs should be a newline. All error

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

\$ cat plaintext1

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

\$ otp\_enc plaintext1 myshortkey 57171 > ciphertext1

\$ 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 \$? \$ cmp plaintext1 plaintext1\_b plaintext1 plaintext1\_b differ: byte 1, line 1 \$ echo \$? \$ 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 \$? \$ **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 "compileali", 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

Start with our sample network programs client.c 👸 & and server.c 👸 &. 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

Sending Data

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

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.

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!

...and then play around with this command:

Finally, I HIGHLY recommend that you develop this program directly on our class server! Doing so will prevent you from having problems transferring the program back and forth, which can cause compatibility issues. Do not use any other non-class server to develop these

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

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