CSE-376 (Spring 2021) Homework Assignment #1

Version 2 (2021-02-10)

Due Saturday, February 27, 2021, 1:00pm ET (13:00 ET)

This assignment is worth 15-20% of your grade, prorated to all assignments and exams we would have (if any).

1. Purpose

To become familiar with the C/Unix development environment and to write a C program that carefully checks for various error conditions. To read and understand documentation about the use of C functions, utility libraries, and system calls.

Expected length of the C program is 500-600 lines of code.

2. Task

Write a C program named "fenc" that secures files by encrypting and ensuring their integrity. Your program must be organized for build with a Makefile. You will have to show that your code runs on the two identical systems with DNS names:

1. cse376u18.cs.stonybrook.edu
2. cse376b.cs.stonybrook.edu

We are going to check out your code from GIT. Then we will use that code to unpack, build, and test your program. We will try to build and run your program by simply typing 'make' in your project directory, on (any machine) like so:

```
$ <checkout your code from git>
$ cd hw1-USER
$ make
$ ./fenc
```

If the above fails, then you may not receive full credit for being able to build on that machine. The "fenc" program has the following usage:

```
fenc [-devh] [-D DBGVAL] [-p PASSFILE] infile outfile
```

The program will prompt the user to enter a password, then read infile, and then produce outfile. If the -e option is given, the program should encrypt infile into outfile, using the supplied user password. If the -d option

is given, the reverse should happen: decrypt infile into outfile, using the supplied password. Either the -d or -e options (but not both) must be supplied (exactly once). Of course, if you use the same password to encrypt and then decrypt a file, you should get back the same exact file data you started with.

For this assignment, you should use getpass(3) by default to input a password from the user. However, as an optional method, you should support -p ARG where "ARG" is the name of a file that contains the password as the first line of the file (this will simplify grading for us). Note that if you read a password from PASSFILE, you should ignore any trailing \n char.

If the -h (help) option is given, the program should print a simple usage line on stderr and exit with a non-zero status code. The usage/help output should also be printed on the console if the wrong number or types of arguments are given to the program.

If the -v option is given, the program should print the version string of the program (which you can assign manually as a #define).

If the -D option is given, the program should print debugging information to stderr based on DBGVAL (a bitmask integer):

- Debug value 0x00 (0d): no debug info printed
- Debug value 0x01 (1d): print a message on immediate entry and right before the exit to every function in your code that you write, including main(), but not library calls you call. Print the name of the function and whether you're entering or exiting it.
- Debug value 0x02 (2d): print right before and right after calling any library call (e.g., from libc, libssl, etc.). Print the function name and whether you're before or after calling it.
- Debug value 0x04 (4d): print right before and right after calling any system call (e.g., open, read, write, close). Print the syscall name and whether you're before or after calling it.
- Debug value 0x10 (16d): print also arguments to any function for debug values 0x1 (upon entry), 0x2 (right before), and 0x4 (right before).
- Debug value 0x20 (32d): print also return values (and errors if any) for any function for debug values 0x1 (right before return), 0x2 (right after), and 0x4 (right after).

Note that you can combine these bits to select subsets of debug info to be printed. For example, the logical OR of 0x01 (1 decimal), 0x10 (16d), and 0x20 (32d) is 0x31 (1+16+32=49d). So if DBGVAL is 49d, it means to enable the combined functionality of 0x01+0x10+0x20.

All options should be processed using getopt(3).

The file names given as input can be any kind of files: relative pathnames or absolute ones. The input file must exist before the program can succeed. The output file may or may not exist: if it exists, it's NOT OK to overwrite it: you have to preserve the output file in case your program fails mid-way, and restore the orig output file upon failure of your program; you can remove the orig file ONLY if your program succeeded entirely. Also be careful in the corner case where your infile and outfile may be the same file (but perhaps named differently).

As special names, if infile is "-", you should read from stdin. If outfile is "-", you should output to stdout. That way, you can use the fenc program as a "Unix filter" such as

```
$ cat foo.in | ./fenc -d - foo.out
```
or
```
$ ./fenc -e foo.in - > foo.out
```

Note that using '-' for infile may conflict with getpass(3), as both want to read from stdin (you'll have to detect this condition and handle it as best you can).

Ideally, the program should check for EVERY POSSIBLE ERROR that could occur BEFORE opening the input and output files and encrypting/decrypting the data. This way you can alert users of problems earlier on. In the case of a possible error, the program should print a detailed error message on stderr and exit with a non-zero status code. This means that you should, for example, check that the input files are readable, that you have permission to read and/or write them (as needed), that neither is a directory, character/block special device, that you have enough disk space to write the output file, didn't run out of quota, etc. Other errors could occur, and it is your job to write the code that checks for all of those conditions. (A good way to think about it is to inspect manual pages for the system calls that you use, and think to yourself "what else could go wrong when this function runs"). Note that some error conditions could be complex to handle: running out of space/quota, or when the two file arguments use different names but actually point to the same file (do not try to encrypt/decrypt a file in place -- this may corrupt the file).

Many of these preconditions can be checked using the l/stat(2) and statfs(2) system calls. You can use the man program to find out programming details about these system calls. For example "man 2 lstat."

The program should copy data efficiently, using read(2) and write(2). It should use the native OS's page size (see the l/stat() and getpagesize(2) functions) as the copy unit. That means, you should allocate such a unit of memory, read the data from infile in such a size, run encrypt/decrypt on it, and then write it out in one such chunk. Note that the end of the file will not always be a multiple of the native file system block size, so you're going to need to handle the copying of the last chunk of the file carefully.

You should check return codes from all system calls that you call, such as read(), write(), open(), and close(), and handle them correctly. In particular, you should carefully handle partial reads or partial writes. You should also handle copying a zero-length file (meaning produce a valid zero-length output file, even though there's no data to encrypt/decrypt). You may NOT use regular libc functions such as fopen, fclose, fgets, or scanf. Use system calls only to check files, open, read, write, and close files (exceptions include fprintf/perror, malloc, free, str*, mem*, etc.).

In general, if you're not sure how "fenc" should behave, check what /bin/cp does: after all, fenc is just like /bin/cp but with encryption/decryption in between.

Your code should be clean, well documented, properly styled and indented.

3. Supporting Encryption and Integrity

Before we get into details, I advise you to HOLD OFF on any encryption stuff until you get the rest of the assignment working. First get the program to read/write, copy a file, handle many flags, handle errors, etc. Only then, start to add encryption support.

You don't need to be an expert in security or encryption to do this assignment. Part of what this assignment will teach you is how to work with someone else's code. So study the openssl library that's installed on the Unix system for this assignment: check man pages, header files, etc. You can also find documentation on openssl.org.

Study how to call openssl functions to encrypt or decrypt a data buffer. Generally you need to select an encryption algorithm, called a cipher: use the Advanced Encryption Standard (AES) for this assignment. You will need a "key" to encrypt the file with. For that, use the password the user supplied; alternatively, you can use openssl to generate a unique key from the password itself, using any one-way function. You also need to select an encryption mode: use the "counter" (CTR) mode. You may also have to set an Initialization Vector (IV): for this assignment, set it to 0 each time you have to set it. (I'll explain these terms in class.)

AES is a "symmetric" cipher, meaning the same key K is used to encrypt and decrypt. Encryption is generally a series of bitwise operations including XOR, AND, OR, and shifts. As such, it's just a series of math operations that cannot fail, meaning there's no way to detect if you're decrypting with the same key used for encryption --- that would result in corrupting the output file (you won't get back the same original file).

To prevent this, we're going to prepend a HASH OF THE KEY inside the file itself. Clearly we don't want to store the key itself in the file, as that reveals a major secret!

A hash is a one-way function that takes input X of any length and produces an output H of a fixed length. Use the SHA256 hashing function, which produces a 256-bit (32B) hash. The key properties of hash functions is that (1) they are one-way, (2) they produce a unique output with very low "collision" probability, and (3) they are non-invertible: given a hash H, it's exceedingly hard to find the original value X. (I'll discuss hash functions as well in class.)

So, before you start to encrypt infile and write outfile, calculate a SHA256 hash of your encryption key K, and store it in the outfile first; then proceed to write the encrypted output. Upon decryption, first read the SHA256 hash (H1) from the start of the file, then compute the same hash for the key K given to the program for decryption (H2), then compare the two hashes: if H1 and H2 match, continue to decrypt; otherwise it means the user has given a different decryption key than was used during encryption, and you should abort with an error.

4. Regression Tests

Create a series of test /bin/sh shell scripts to test that your program runs as expected. Name them test1.sh, test2.sh, etc.

1. One test would verify that your program handles command line interface (CLI) options correctly. This includes testing for illegal/missing CLI options (e.g., -d and -e can't be given together).
2. Test to ensure that you preserve the output file exactly as it was before, in case a failure happened mid-way in reading the infile or writing the outfile.
3. Test the code that verifies that the decryption key is the same as the encryption key.

5. Makefile

Create a Makefile to build the program, any intermediate binaries, and set dependencies on any header file, library, etc.

- Use gcc by default but don't hard code it, use the CC variable.
- Turn on compile flags -g -O2 -Wall -Werror, using CFLAGS.
- set LDFLAGS to any libraries needed.
- recompile the sources into individual .o files ONLY if the corresponding .c file had changed.
- [re]link the final executable "fenc" only if any of the corresponding
- .o files (or libraries) had changed.
- recompile any of the .c files if any of their dependent .h files have changed as well.
- add a "make clean" target to clean all regenerable files
- add a "make tests" target to run the series of regression tests.

6. Style and More

Aside from testing the proper functionality of your code, we will also evaluate the quality of your code. Be sure to use a consistent style, well documented, and break your code into separate functions and/or source files as it makes sense.

To be sure your code is very clean, it must compile with "gcc -Wall -Werror" without any errors or warnings! If the various sources you use require common definitions, then do not duplicate the definitions. Make use of C's code-sharing facilities.

You must include a README file with this and any assignment. The README file should describe what you did, what approach you took, results of any measurements you made, which files are included in your submission and what they are for, etc. Feel free to include any other information you think is helpful to us in this README; it can only help your grade. The code you write should be your own, but if you want to use any online code, you must clear it with me (note: github sources NOT allowed), and cite it both in your code and your README.

7. Submission

You will need to submit all of your sources, headers, scripts, Makefiles, and README. Submission is accepted via GIT only! Do not submit regenerable files like binaries, *.o files, or any temp files -- only true "source" files.

PLEASE test your submission before submitting it, by checking it out in a separate directory, compiling it cleanly, and testing it again. DO NOT make the common mistake of writing code until the very last minute, and then trying to figure out how to use GIT and skipping the testing of what you submitted. You will lose valuable points if you do not get to submit on time or if your submission is incomplete!!!

(General GIT submission guidelines are available on the class Website.)

8. Extra Credit

If you do any of the extra credit work, then your EC code must be wrapped in

```
#ifdef EXTRA_CREDIT
     // EC code here
#else
     // base assignment code here
#endif
```

This extra credit is worth a total of 14 extra points (the main assignment is worth 100 points).

8.1. [A] 3 points.

Add an option called -s (safe) which will cause the program to prompt for the user password twice (not just once), to ensure that the password was typed correctly. If the passwords didn't match, abort with an appropriate error.

8.2. [B] 3 points.

Add another option -i which will utilize the Initialization Vector (IV) part of the fenc. Without having to know much about the IV, it is useful to understand that setting it to a different value each time you encrypt or decrypt a chunk of bytes, produces stronger encryption that is harder to break. A common way to set the IV is as follows: use the index of the page (or page number) that you are encrypting or decrypting (e.g., on an i386 system with a 4096-byte page size, bytes 0-4095 are in page 0, bytes 4096-8191 are in page 1, etc.).

8.3. [C] 4 points.

Add another option -c ARG to the fenc program. This option, when used, will tell the program to use the "ARG" symmetric cipher instead of AES (e.g., blowfish, DES, etc.). If a cipher doesn't exist, flag an error (you can find if it exists in libssl). If you use a different cipher, you'll need to store the cipher's name in the file header, and ensure that you only decrypt with the same cipher (just as you store a hash of the key K).

8.4. [D] 4 points (max).

To incentivize you to submit your code earlier than the deadline, we'll give you 2 points if you submit your assignment at least 24 hours before the official deadline; and 4 points if you submit at least 48 hours before. Note that we count the LAST git-push to your repo as the last time you submitted the assignment, even if you made a very small change.

Good luck.

9. Copyright and Academic Integrity Statement

(c) 2021 Erez Zadok
(c) Stony Brook University

DO NOT POST ANY PART OF THIS ASSIGNMENT OR MATERIALS FROM THIS COURSE ONLINE IN ANY PUBLIC FORUM, WEB SITE, BLOG, ETC. DO NOT POST ANY OF YOUR CODE, SOLUTIONS, NOTES, ETC.

ALL CODE SHOULD BE YOURS! WE WILL COMPARE YOUR CODE USING SEMI-AUTOMATED CODE-COMPARISON TOOLS AGAINST CODE USED IN PAST YEARS IN THIS CLASS AS WELL AS CODE WE MAY DOWNLOAD FROM THE INTERNET. ALL SUSPECTED PLAGIARISM OR CHEATING CASES WILL BE INVESTIGATED AND POSSIBLY REFERRED FURTHER.

10. ChangeLog

- v1: initial draft.
- v2: reviewed by TA.