Assignment 2: a file encryptor

> NOTE:
>
> An assignment overview video is available here.

Contents

- Aims
- Introduction
- Getting Started
  - Reference Implementation
- Your Tasks
- Subset 0: File and directory commands
- Subset 1: XOR encryption
- Subset 2: Directory traversal
- Subset 3: ECB encryption
- Subset 4: Cipher block chaining encryption
- Testing
- Assumptions and Clarifications
- Assessment
  - Testing
  - Submission
  - Due Date
  - Assessment Scheme
  - Intermediate Versions of Work
  - Assignment Conditions
- Change Log

Contents

Aims

Introduction

Getting Started

tide Examples

Your Tasks

Subset 0: File and directory commands

Subset 1: XOR encryption

Subset 2: Directory Traversal

Subset 3: Electronic Codebook

Subset 4: Cipher Block Chaining (Challenge!)

Testing

Assumptions and Clarifications

Assessment

Aims

- to improve your understanding of filesystem objects
- to give you experience writing C code to manipulate binary files
- to further experience practical uses of bitwise operations
- to give you experience writing a relevant low-level data manipulation program in C

Introduction

Your task in this assignment is to write **tide**, a terribly insecure single-file encryption/decryption tool. Throughout this assignment, you will explore some basic filesystem operations, as well as implement several rudimentary encryption algorithms.

Encryption is the process of converting information into an obscured format, which can (in theory), only be converted back into useful information by an authorized party who knows the encryption process and key. Encryption is an incredibly useful tool, and is the reason why the internet can function in the way it does, with sensitive information freely transmitted across it.

File encryption is particularly useful to safeguard data in the case that it is stolen. Encrypting your files could prevent someone from being able to access your photos in the event that your laptop gets stolen.

In this assignment, you will implement three different algorithms for file encryption: XOR (eXclusive OR), ECB (Electronic Code Book) and CBC (Cipher Block Chaining). Each of these algorithms function slightly differently, but all work towards the same purpose of obscuring information, that can only be correctly interpreted by an authorised party.

XOR encryption works by employing the bitwise XOR operation on every bit of some given data. A key, which when broken up into it's constituent bits, expanded to much the length of the data being encrypted. The XOR operation is then employed between these two bitstreams to yield the encrypted data. This encrypted data can be decrypted only by re-running the same XOR operation with the same key. In tide, standalone XOR encryption will only employ the the single-byte key `0xA9`.

ECB encryption works by bit-shifting data by the amount specified by some key (a password). Each character in a 'block' of the input data is shifted by the value of the character in the corresponding position within the password. The encrypted data can be decrypted only by shifting it back by the value of the corresponding position within the password. In tide, passwords

will be a fixed length of 16 characters.

CBC encryption is different from the above two algorithms as each block of the encrypted data contributes to the encryption of the next block. We will combine both XOR encryption and ECB encryption to develop an encryption algorithm where it is significantly harder for an unauthorised party to read our encrypted data by guessing our password.

However, before all of this, tide needs to be able to function as a basic standalone program. As such, we will implement several filesystem manipulation operations. You will also implement two different methods of searching for files, which will make the user's life easier in finding what they might need to encrypt.

Getting Started

Create a new directory for this assignment called `tide`, change to this directory, and fetch the provided code by running these commands:

```
$ mkdir -m 700 tide
$ cd tide
$ 1521 fetch tide
```

If you're not working at CSE, you can download the provided files as a zip file or a tar file.

This will get you `tide.c`, which contains code to start the assignment. As provided, it will compile and run, but lacks any real functionality:

```
$ make
dcc -Wall -Werror main.c tide.c -o tide
$ ./tide
Welcome to tide!
To see what commands are available, type help.
tide> help
help (h)                          Prints this help message
pwd (p)                           Prints the current directory
chdir directory (cd)              Changes the current directory
list (ls)                         Lists the contents of the current directory
test-encryptable filename (t)     Tests if a file can be encrypted
xor-contents filename (x)         Encrypts a file with simple XOR
encrypt-ecb filename (ee)         Encrypts a file with ECB
decrypt-ecb filename (de)         Decrypts a file with ECB
search-name search-term (sn)      Searches for a file by filename
search-content search-size (sc)   Searches for a file by its content for the provided bytes
search-from-file source-file (sf) Searches for a file by its content for the provided bytes, supplied
from a file
encrypt-cbc filename (ec)         Encrypts a file with CBC
decrypt-cbc filename (dc)         Decrypts a file with CBC
quit (q)                          Quits the program
tide> q
Thanks for using tide. Have a nice day!
```


However, `tide.c` also contains some provided functions to make your task easier. For example, the `sort_strings` function will sort an array of strings into alphabetical order in-place. You should read through the provided code in this file before you begin work on this assignment.

You may also find the provided constants, data types and function signatures in `tide.h` to be useful.

Reference implementation

A reference implementation is a common, efficient, and effective method to provide or define an operational specification; and it's something you will likely work with after you leave UNSW.

We've provided a reference implementation, `1521 tide`, which you can use to find the correct outputs and behaviours for any input:

```
$ 1521 tide
Welcome to tide!
To see what commands are available, type help.
tide> help
help (h)                          Prints this help message
pwd (p)                           Prints the current directory
chdir directory (cd)              Changes the current directory
list (ls)                         Lists the contents of the current directory
test-encryptable filename (t)     Tests if a file can be encrypted
xor-contents filename (x)         Encrypts a file with simple XOR
encrypt-ecb filename (ee)         Encrypts a file with ECB
decrypt-ecb filename (de)         Decrypts a file with ECB
search-name search-term (sn)      Searches for a file by filename
search-content search-size (sc)   Searches for a file by its content for the provided bytes
search-from-file source-file (sf) Searches for a file by its content for the provided bytes, supplied
from a file
encrypt-cbc filename (ec)         Encrypts a file with CBC
decrypt-cbc filename (dc)         Decrypts a file with CBC
quit (q)                          Quits the program
tide> q
Thanks for using tide. Have a nice day!
```

tide also has a **colored** mode, to make it a bit nicer to use.

```
$ 1521 tide --colors
Welcome to tide!
To see what commands are available, type help.
tide> q
Thanks for using tide. Have a nice day!
```

(as tends to be the convention in computing, we use the Americanised spelling of colour here.)

Every concrete example shown below is runnable using the reference implementation. Your goal is to make your program run just as the reference implementation does, with identical output and behaviour.

Where any aspect of this assignment is undefined in this specification, you should match the behaviour exhibited by the reference implementation. Discovering and matching the reference implementation's behaviour is deliberately a part of this assignment.

If you discover what you believe to be a bug in the reference implementation, please report it in the course forum. If it is a bug, we may fix the bug; or otherwise indicate that you do not need to match the reference implementation's behaviour in that specific case.

tide Examples

Additionally provided for your use is a command `1521 tide-examples`.

When executed, it will create an `examples` directory in the current directory, and will create a number of files intended for testing, in this directory. An example of its usage follows:

```
$ ls
commands.h  escape.h  main.c  tide.c  tide.h  Makefile
$ 1521 tide-examples
$ ls
commands.h  escape.h  main.c  tide.c  tide.h  Makefile
$ ls examples
a  empty.txt  forbidden  lorem  lorem.txt  ro_dir  tide_sols.txt
```

The autotests make heavy use of these examples, so it is recommended to run this command before manually replicating any autotest output you intend to debug.

Your Tasks

This assignment consists of five subsets. Each subset builds on the work of the previous one, and each subset is more complex than the previous one.

It is recommended that you work on each subset in order.

Subset 0 | Subset 1 | Subset 2 | Subset 3 | Subset 4

Subset 0: File and directory commands

RATIONALE:

Why does this subset exist? This subset tests your ability to interact with low-level C library calls and system calls to interact with file system objects, and to give you a better idea of detecting and handling errors with these calls.

Who do we expect to complete this subset? We hope that all students will be able to complete this subset.

What help will tutors provide to students? Tutors can give you a theoretical explanation, and give lots of help with the lab exercises that lead to this assignment. Tutors will be happy to help you debug any issues with your code.

How many marks will I get? Completing this subset, with good style, is worth around 50% of the performance component of this assignment.

For this subset, you will need to implement the following three functions:

- `void print_current_directory()`
- `void change_directory(char *directory)`
- `void list_current_directory()`

All user input is handled for you in `main.c`. You will never have to read input from `stdin` - all inputs are passed in through arguments to these functions for you.

Printing the current directory

You will need to implement `print_current_directory`, such that it prints the current directory the program is operating in.

Once you have this function working correctly, your tide implementation should match the following behaviour:

```
$ 1521 tide-examples
$ cd examples
$ 1521 tide
Welcome to tide!
To see what commands are available, type help.

tide> p
The current directory is: /home/z5555555/tide/examples
tide> q
Thanks for using tide. Have a nice day!
$ cd a
$ 1521 tide
Welcome to tide!
To see what commands are available, type help.

tide> p
The current directory is: /home/z5555555/tide/examples/a
tide> q
Thanks for using tide. Have a nice day!
```


(Your home directory will, of course, feature your zID instead, and your overall path may be a little different!)

HINT:

You may find the lecture example `getcwd.c`, which shows how to print your working directory with *getcwd* useful.

NOTE:

You may assume that *getcwd* will never fail (if used correctly).

You can assume that any path tide handles has a maximum length of 4096. A constant, `MAX_PATH_LEN`, has been provided for you in `tide.h`.

No error checking is necessary.

Changing directories

You will need to implement `change_directory` so that it changes the current working directory of tide. All other functions should now operate on the new working directory.

Supplied directories can be both relative and absolute. Additionally, your implementation should expand `~` into the user's home directory, using the `HOME` environment variable.

Once you have this function working correctly, your tide implementation should match the following behaviour:

```
$ 1521 tide-examples
$ cd examples
$ 1521 tide
Welcome to tide!
To see what commands are available, type help.

tide> cd a
Moving to a
tide> p
The current directory is: /home/z5555555/tide/examples/a
tide> cd ..
Moving to ..
tide> p
The current directory is:  /home/z5555555/tide/examples
tide> cd this_doesnt_exist
Could not change directory.
tide> q
Thanks for using tide. Have a nice day!
```

HINT:

You may find the lecture example `my_cd.c`, which shows how to change directories with *chdir* useful.

Additionally, you may find *getenv* useful in order to retrieve the user's home directory.

NOTE:

If the *chdir* syscall fails, you must print `MSG_ERROR_CHANGE_DIR`.

Special directories such as `.` and `..` should evaluate to the current directory and parent directory respectively - you shouldn't need to do any special handling for these!

Listing the current directory

`list_current_directory` should print every file and folder in tide's working directory, along with its permissions. The output of this function should be sorted.

Fortunately, a sort function has been provided for you! Calling `sort_strings` sorts the supplied array of strings in-place. You can then print the now-sorted array in order, in the required format.

Once you have this function working correctly, your tide implementation should match the following behaviour:

```
$ 1521 tide-examples
$ cd examples
$ 1521 tide
Welcome to tide!
To see what commands are available, type help.
tide> ls
drwxr-xr-x  .
drwxr-xr-x  ..
drwxr-xr-x  a
-rw-r--r--  empty.txt
drwxr-xr-x  forbidden
drwxr-xr-x  lorem
-rw-r--r--  lorem.txt
drwxr-xr-x  ro_dir
-rw-r--r--  tide_sols.txt
tide> q
Thanks for using tide. Have a nice day!
```

(Your `..` directory's permissions may look a little different!)

NOTE:

The permission string is separated from the filename by the tab character (`'\t'`).

You can assume that a directory has a maximum of 512 entries. Your solution does not need to support more than this. A constant, `MAX_LISTINGS`, has been provided to you in `tide.h`.

You can assume the *opendir* function will never fail to open the `.` directory (if used correctly).

No error checking is necessary.

HINT:

You may find the lecture example `list directory.c`, which shows how to list the contents of your current directory with *opendir*, *readdir* and *closedir* useful. The information in *stat* and *inode* will likely also be useful.

HINT:

You may find the lecture example `stat.c`, which shows how to retrieve file attributes with *stat* useful.
You may additionally find the information on *inode*s of use too.

NOTE:

tide is only expected to handle regular files and directories as input.

You must print `MSG_ERROR_FILE_STAT` if your call to *stat* fails. All provided error messages for this assignment should be printed to `stdout`.

Testing this subset

You can test and submit this subset with:

```
$ 1521 autotest tide subset0
$ give cs1521 ass2_tide tide.c [other .c or .h files]
```

Testing

You are expected to do your own testing. Some autotests are available to help you get started:

```
$ 1521 autotest tide
```

You can create extra `.c` or `.h` files; but you will need to supply them explicitly to autotest as follows:

```
$ 1521 autotest tide [other .c or .h files]
```


Assumptions and Clarifications

Like all good programmers, you should make as few assumptions as possible.

- Your submitted code must be written in C. You may not submit code in other languages.
- You should avoid leaking memory wherever possible.
- All supplied error messages should be printed to `stdout`.
- You can call functions from the C standard library available by default on CSE Linux systems: including, e.g., `stdio.h`, `stdlib.h`, `string.h`, `assert.h`.

  Additionally, you may call functions from the C POSIX libraries available on CSE Linux systems: including, e.g., `unistd.h`, `sys/stat.h`, `dirent.h`.
- You may not create subprocesses: you may not use *posix_spawn*, *posix_spawnp*, *system*, *popen*, *fork*, *vfork*, *clone*, or any of the `exec*` family of functions, like *execve*.
- You may not create or use temporary files.
- *tide* only has to handle ordinary files and directories.

  tide does not have to handle symbolic links, devices or other special files.

  tide will not be given directories containing symbolic links, devices or other special files.

  tide does not have to handle hard links.

If you need clarification on what you can and cannot use or do for this assignment, please ask in the course forum.

You are required to submit intermediate versions of your assignment. See below for details.

Your program must not require extra compile options. It must compile with `dcc *.c -o tide`, and it will be run with `dcc` when marking. Run-time errors from illegal C will cause your code to fail automarking.

If your program writes out debugging output, it will fail automarking tests. Make sure you disable debugging output before submission.

Assessment

Testing

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest tide [optionally: any extra .c or .h files]
```

You can also run autotests for a specific subset. For example, to run all tests from subset 0:

```
$ 1521 autotest tide subset0 [optionally: any extra .c or .h files]
```

Some tests are more complex than others. If you are failing more than one test, you are encouraged to focus on solving the first of those failing tests. To do so, you can run a specific test by giving its name to the `autotest` command:

```
$ 1521 autotest tide subset0_test1 [optionally: any extra .c or .h files]
```

`1521 autotest` will not test everything.
Always do your own testing.

Automarking will be run by the lecturer after the submission deadline, using a superset of tests to those `autotest` runs for you.

> WARNING:
>
> Whilst we can detect errors have occurred, it is often substantially harder to automatically explain what that error was. As you continue into later subsets. the errors from `1521 autotest` will become less and less clear or useful. You will need to do your own debugging and analysis.

Submission

When you are finished working on the assignment, you must submit your work by running `give` :

```
$ give cs1521 ass2_tide tide.c [optionally: any extra .c or .h files]
```


You must run `give` before **Week 10 Friday 22:00:00** to obtain the marks for this assignment. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

You can run `give` multiple times.
Only your last submission will be marked.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

You *cannot* obtain marks by emailing your code to tutors or lecturers.

You can check your latest submission on CSE servers with:

```
$ 1521 classrun check ass2_tide
```

You can check the files you have submitted here.

Manual marking will be done by your tutor, who will mark for style and readability, as described in the **Assessment** section below. After your tutor has assessed your work, you can view your results here; The resulting mark will also be available via give's web interface.

Due Date

This assignment is due **Week 10 Friday 22:00:00** (2023-11-17 22:00:00).

The UNSW standard late penalty for assessment is 5% per day for 5 days - this is implemented hourly for this assignment.

Your assignment mark will be reduced by 0.2% for each hour (or part thereof) late past the submission deadline.

For example, if an assignment worth 60% was submitted half an hour late, it would be awarded 59.8%, whereas if it was submitted past 10 hours late, it would be awarded 57.8%.

Beware - submissions 5 or more days late will receive zero marks. This again is the UNSW standard assessment policy.

Assessment Scheme

This assignment will contribute **15** marks to your final COMP1521 mark.

80% of the marks for assignment 2 will come from the performance of your code on a large series of tests.

20% of the marks for assignment 2 will come from hand marking. These marks will be awarded on the basis of clarity, commenting, elegance and style. In other words, you will be assessed on how easy it is for a human to read and understand your program.

An indicative assessment scheme for performance follows. The lecturer may vary the assessment scheme after inspecting the assignment submissions, but it is likely to be broadly similar to the following:

100% for performance	implements all behaviour perfectly, following the spec exactly.
90% for performance	completely working subsets[0-3].
80% for performance	completely working subsets[0-2].
65% for performance	completely working subsets[0-1].
50% for performance	completely working subset0.
30-40% for performance	good progress, but not passing subset0 autotests.
0%	knowingly providing your work to anyone and it is subsequently submitted (by anyone).
0 FL for COMP1521	submitting any other person's work; this includes joint work.
academic misconduct	submitting another person's work without their consent; paying another person to do work for you.

An indicative assessment scheme for style follows. The lecturer may vary the assessment scheme after inspecting the assignment submissions, but it is likely to be broadly similar to the following:

100% for style	perfect style
90% for style	great style, almost all style characteristics perfect.
80% for style	good style, one or two style characteristics not well done.
70% for style	good style, a few style characteristics not well done.
60% for style	ok style, an attempt at most style characteristics.
≤ 50% for style	an attempt at style.

An indicative style rubric follows:

- Formatting **(6/20)**:
  - Whitespace (e.g. `1 + 2` instead of `1+2`)
  - Indentation (consistent, tabs or spaces are okay)
  - Line length (below 100 characters unless very exceptional)
  - Line breaks (using vertical whitespace to improve readability)
- Documentation **(8/20)**:
  - Header comment (with name and zID)
  - Function comments (above each function with a description)
  - Descriptive variable names (e.g. `char *home_directory` instead of `char *h`)
  - Descriptive function names (e.g. `get_home_directory` instead of `get_hd`)
  - Sensible commenting throughout the code (don't comment every single line; leave comments when necessary)
- Elegance **(5/20)**:
  - Does this code avoid redundancy? (e.g. Don't repeat yourself!)
  - Are helper functions used to reduce complexity? (functions should be small and simple where possible)
  - Are constants appropriately created and used? (magic numbers should be avoided)
- Portability **(1/20)**:
  - Would this code be able to compile and behave as expected on other POSIX-compliant machines? (using standard libraries without platform-specific code)
  - Does this code make any assumptions about the endianness of the machine it is running on?

Note that the following penalties apply to your total mark for plagiarism:

0 for asst2	knowingly providing your work to anyone and it is subsequently submitted (by anyone).
0 FL for COMP1521	submitting any other person's work; this includes joint work.
academic misconduct	submitting another person's work without their consent; paying another person to do work for you.

Intermediate Versions of Work

You are required to submit intermediate versions of your assignment.

Every time you work on the assignment and make some progress you should copy your work to your CSE account and submit it using the `give` command below. It is fine if intermediate versions do not compile or otherwise fail submission tests. Only the final submitted version of your assignment will be marked.

Assignment Conditions

- **Joint work** is **not permitted** on this assignment.

  This is an individual assignment. The work you submit must be entirely your own work: submission of work even partly written by any other person is not permitted.

  Do not request help from anyone other than the teaching staff of COMP1521 — for example, in the course forum, or in help sessions.

  Do not post your assignment code to the course forum. The teaching staff can view code you have recently submitted with give, or recently autotested.

  Assignment submissions are routinely examined both automatically and manually for work written by others.

  Rationale: this assignment is designed to develop the individual skills needed to produce an entire working program. Using code written by, or taken from, other people will stop you learning these skills. Other CSE courses focus on skills needed for working in a team.

- The use of generative tools such as Github Copilot, ChatGPT, Google Bard is **not permitted** on this assignment.

  Rationale: this assignment is designed to develop your understanding of basic concepts. Using synthesis tools will stop you learning these fundamental concepts, which will significantly impact your ability to complete future courses.

- **Sharing, publishing, or distributing** your assignment work is **not permitted**.

  Do not provide or show your assignment work to any other person, other than the teaching staff of COMP1521. For example, do not message your work to friends.

  Do not publish your assignment code via the Internet. For example, do not place your assignment in a public GitHub repository.

  Rationale: by publishing or sharing your work, you are facilitating other students using your work. If other students find your assignment work and submit part or all of it as their own work, you may become involved in an academic integrity investigation.

- **Sharing, publishing, or distributing** your assignment work after the completion of COMP1521 is **not permitted**.

  For example, do not place your assignment in a public GitHub repository after this offering of COMP1521 is over.

  Rationale: COMP1521 may reuse assignment themes covering similar concepts and content. If students in future terms find your assignment work and submit part or all of it as their own work, you may become involved in an academic integrity investigation.

Violation of any of the above conditions may result in an academic integrity investigation, with possible penalties up to and including a mark of 0 in COMP1521, and exclusion from future studies at UNSW. For more information, read the UNSW Student Code, or contact the course account.