COMP3301 2022 Assignment 1 - USB Encryption Device

- Due: Week 5
- $Revision: 350 $

1 USB Encryption Device

This assignment will develop a program which controls a USB device used to encrypt and decrypt text messages.

You will be provided with partially-working code, to which you will need to make fixes and improvements. The purpose of this assignment is to demonstrate your skills in reading and understanding existing code as well as fixing and improving it, while exposing you to the USB device model and many OpenBSD-specific APIs and tools.

This is an individual assignment. You should feel free to discuss aspects of C programming and the assignment specification with fellow students, and discuss OpenBSD and its APIs in general terms. You should not actively help (or seek help from) other students with either the correct diagnosis of the coding errors in task 1, or the actual coding of your improvements for the rest of the assignment. It is cheating to look at another student's code and it is cheating to allow your code to be seen or shared in printed or electronic form. You should note that all submitted code will be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion (outside of the base code given to everyone), formal misconduct proceedings will be initiated against you. If you're having trouble, seek help from a member of the teaching staff. Don't be tempted to copy another student's code. You should read and understand the statements on student misconduct in the course profile and on the school web-site: https://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism

2 Background

The A1 USB device is a more complex version of the "dummy" device used in Prac 3. In Prac 3 you saw a very simple device that accepts data, "encrypts" it, and then lets you read the result back in.

For this assignment, the device has been extended. It now supports:

- Encryption and decryption requests up to 64k in length
- Multiple concurrent operations
- Asynchronous notification of operation progress and completion

The full device interface specification is available at:

https://stluc.manta.uqcloud.net/comp3301/public/2022/a1-device-spec.pdf

This document may be used as a reference for all the details of how this new device communicates with the host over USB. Note that you will not need to implement the host side of this from scratch: you are being provided with a partly-working implementation to fix and modify.

The host software you will be working on consists of two programs: a daemon (`usbcryptd`) and a command-line client tool (`usbcrypt`).

The daemon is a persistent background program which accepts requests from the command-line client and issues them to hardware. Multiple instances of the client tool may be run at once, but only one daemon will be run at a time. The daemon will issue the requests given to it by clients to the hardware device in parallel if possible, or else it will wait until resources are available.

Each run of the command-line client tool should issue a single encryption or decryption request. What kind of operation is requested and the data to be processed are specified using command-line arguments, as explained in the specification below.

The command-line tool and the daemon communicate with each other via connection-oriented UNIX domain sockets. These are similar to a a TCP socket, and use the same functions and API, which you should already have experience using from CSSE2310. The key difference is that UNIX domain sockets have a node on the filesystem rather than an IP address and port, so connections can only be made within the same computer.

In CSSE2310 you learned how to write socket servers using blocking system calls, making use of threads for concurrency to allow the server to handle multiple requests. The daemon for this assignment uses a thread-based approach for concurrency, similar to what you should have already seen.

To communicate with the USB bus, the daemon uses OpenBSD's `ugen(4)` device driver. `ugen` allows userland programs to control USB devices which have no kernel driver. In Prac 3 you covered the basics of using `ugen` and how it works.

3 Specifications

This assignment consists of two parts:

1. Diagnosing and fixing an error in the existing program; and
2. Adding support for a new feature: progress updates during an encryption or decryption operation, which are relayed to the commandline client and displayed to the user.

Part 1 will produce both a non-code artifact (a short summary of how you found the error and diagnosed it) and code (a fix for the error). Both of these will need to be checked into your Git repository in accordance with the instructions below. Part 2 will produce only code.

3.1 Part 1: Diagnosing and fixing an error

The code you have been provided includes several errors or bugs. In this part, you will find and locate a major error related to the handling of concurrent requests. This error is located in the daemon (`usbcryptd`) code.

Have a think through the different ways in which you can test the provided code to discover the error, and then how you can debug it.

Keep track of the steps required to produce the error and all the facts that you can gather about it. Then write an explanation of why you think the error has occurred and what can be done about it.

Your reproduction steps and analysis should be checked into your source repository in the text file `/usr/src/a1-error.txt` (or `/a1-error.txt` relative to the repository root). A suggested template for how to format this file is included in this specification as Appendix A.

Once you have diagnosed and analysed the error, you should fix it. Your fix should be committed to your repository as a separate commit to the one importing the base code for this assignment.

Your fix does not need to be a minimal fix to be marked correct, but we would advise that we do not expect this activity to take a lot of time to complete once you have found and diagnosed the error.

If you find more than one issue, write about the most severe one (e.g. the one that crashes the program or results in the most clearly incorrect behaviour). We may accept multiple correct answers for Part 1, as long as they are supported by evidence and clearly explained.

3.2 Part 2: Adding progress updates

Next, you will add a new feature to the codebase provided: progress updates for ongoing operations.

The hardware device supports reporting when an ongoing operation reaches 25%, 50% and 75% of completion. You will need to find the appropriate place in the daemon code which communicates with the hardware to gather this information, and then you will need to relay it back to the commandline client tool.

The commandline client tool will also need to be modified to display this progress information to the user.

You must choose how to re-design the format of data carried on the socket connection between the client tool and the daemon to achieve this, and marks are allocated for using a design that is clear and understandable from the code and any comments or documentation you add. You should also consider how to make sure that progress updates cannot be confused with the actual content of the data by the client.

3.3 Command-line interfaces

3.3.1 Daemon

The daemon should have the following command-line interface:

```
Usage: usbcryptd [options]

Options supported:
  -f        Run in foreground; don't daemonise
  -v        Enable verbose output on stderr (implies -f)
  -p PATH   Path to the UNIX socket to listen on
            (if not specified, uses /var/run/usbcryptd.sock)
```

When run without `-f` or `-v`, the `usbcryptd` command will daemonize using daemon(3).

If the UNIX socket `usbcryptd` is set to listen on already exists, it will delete (unlink) it before establishing its own. The UNIX socket will be created and bound before daemonising, and if errors are encountered doing so, `usbcryptd` should exit with exit status 1 and print an error message to stderr in the style of `err(3)`.

`usbcryptd` should also check for the presence of the A1 device on the system before daemonising. If the device is not present, or an error occurs enumerating USB devices, it should exit with exit status 1 and print an error message in the style of `err(3)`.

3.3.2 Client

The client should have the following command-line interface:

```
Usage: usbcrypt [options] [PATH]

Arguments:
  PATH      Path to an input file to process
            If not specified, use stdin

Options supported:
 -o PATH    Output to a file rather than stdout
 -d         Decrypt rather than encrypt (the default)
 -q         Quiet; don't display progress or errors on stderr
 -p PATH    Path to the usbcryptd UNIX socket
            (if not specified, uses /var/run/usbcryptd.sock)
```

The client should wait (block) until the request has been processed and a response is received. If an error response is received, it should exit with non-zero exit status, and display an error message (unless `-q` is given). If another kind of fatal error occurs (e.g. a socket error or invalid format of data), it should exit with non-zero exit status and display an error message in the format of `err(3)`, ignoring `-q`.

An example of using the client (after progress display has been implemented in part 2):

```
$ echo 'hello world' | usbcrypt
Progress: 25%......50%......75%.......100%
uryyb jbeyq
```

```
$ echo 'uryyb jbeyq' | usbcrypt -dq -o file
$ cat file
hello world
```

The progress display (entire line of output beginning with "Progress:") should be output to stderr. Error messages should also be emitted to stderr. Only the encrypted or decrypted data itself should be written to stdout (if the `-o` option is not given).

3.4 Writing Style

In part 1, you will need to produce a written summary of how you found and diagnosed the error. This is to be written in English, with appropriate grammar, punctuation and style for general business writing (think roughly: "would I write this to my supervisor at work?"). You will not lose marks for trivial spelling or grammatical errors which do not obscure the meaning of what you have written, but we recommend taking some care.

3.5 Code Style

Your code is to be written according to OpenBSD's style guide, as per the `style(9)` man page.

An automatic tool for checking for style violations is available at https://stluc.manta.uqcloud.net/comp3301/public/2022/cstyle.pl. This tool will be used for calculating your style marks for this assignment.

3.6 Compilation

Your code for this assignment is to be built on an `amd64` OpenBSD 7.1 system identical to your course-provided VM. It must compile as a result of running `make(1)` in the `/usr/src/usr.sbin/usbcryptd` and `/usr/src/usr.bin/usbcrypt` directories (or `usr.sbin/usbcryptd` and `usr.bin/usbcrypt` relative to your repository root).

The existing `Makefile`s in the provided code are functional as-is, but may need modification as part of your work for this assignment.

Compilation must produce binaries named `usbcryptd` and `usbcrypt`, in the same manner as with the provided `Makefile`, using `bsd.prog.mk`.

Note that the existing Makefile ensures the `-Wall` flag is passed to the compiler, as well as a few other warning and error-related flags. We will be compiling your code with these flags forced on, so you should not remove or modify them.

3.7 Provided code

The provided code which forms the basis for this assignment can be downloaded as a single patch file at:

https://stluc.manta.uqcloud.net/comp3301/public/2022/a1-base.patch

You should create a new `a1` branch in your repository based on the `openbsd-7.1` tag using `git checkout`, and then apply this base patch using the `git am` command:

```
$ git checkout -b a1 openbsd-7.1
$ ftp https://stluc.manta.uqcloud.net/comp3301/public/2022/a1-base.patch
$ git am < a1-base.patch
$ git push origin a1
```

3.8 Recommendations

The focus of this assignment is reading and modifying existing code, as well as understanding the USB device model. We don't expect you to "reinvent the wheel" if there is common functionality available in OpenBSD already which you can leverage.

It is strongly recommended that the following APIs are used as part of your program:

- The BSD sockets API:
  - `socket(2)`
  - `connect(2)`
  - `bind(2)`
  - `accept(2)`
  - `unix(4)` socket family
- `ugen(4)`
- `ioctl(2)`
- `pthreads(3)`
- `getopt(3)`
- `err(3)`

Most of these are already in use by the provided code.

4 Submission

Submission must be made electronically by committing to your Git repository on `source.eait.uq.edu.au`. In order to mark your assignment the markers will check out the `a1` branch from your repository. Code checked in to any other branch in your repository will not be marked.

As per the `source.eait.uq.edu.au` usage guidelines, you should only commit source code and Makefiles.

Your `a1` branch should consist of:

- The openbsd-7.1 base commit
- The A1 base patch commit
- Commit(s) for Part 1
- Commit(s) for Part 2

We expect to see separate commits for parts 1 and 2. If you do not submit separate commits for the two parts, we will endeavour to mark your work as best we can, but it may take longer and we may not be able to discern as effectively how to assign partial credit for incomplete work.

4.1 Marking

Your submission will be marked by course tutors and staff, who will annotate feedback on your code using an online tool (Reviewboard).

You will receive an e-mail containing a link to your detailed feedback when it is available, as well as a final grade on Blackboard.

5 Testing

We will be testing your code's functionality by compiling and running it in a virtual machine set up in the same way we detail in Prac 1.

Your `usbcrypt` and `usbcryptd` programs will be tested together, not separately, so you are free to change the socket interface between them.

Testing will consist of pre-prepared scripts that will produce output that will be inspected by course staff. Small differences in the output format of commands (e.g. "usage" messages or the progress display) are acceptable and will not result in lost marks unless they obscure the meaning of the message or status of the program. However, command-line options, the characters assigned to them, exit status numbers, and their meaning must be implemented exactly as specified.

As well as testing your code by running commands within your VM, you may also adjust some of the behaviour of the simulated device used in this assignment.

The VM control interface will have several commands added for adjusting aspects of the device's behaviour and viewing some basic statistics about its operation. Please see the VM control interface documentation (once updated for A1):

https://stluc.manta.uqcloud.net/comp3301/public/2022/vm-control.pdf

6 Appendix A: Bug Report Template

The following is the format in which you should provide your bug report for Part 1:

```
Summary
-------
A short one-sentence summary of the issue.

Steps to reproduce
------------------

1. ...
2. ...

Analysis
--------

Paragraphs of text. This should be a narrative which describes how the bug
was first found, what analysis was performed to determine the cause of the bug,
showing the evidence along the way.

Include evidence as indented blocks of command output or code.

Proposed fix
------------

Summarise the changes that need to be made to the code and why they will fix
the problem.
```