COMP3301 2022 Assignment 2

- Due: 23rd of September 2022, Week 9
- $Revision: 385 $

1 OpenBSD `ccid(4)` USB CCID Driver

The task in this assignment is to develop an OpenBSD kernel driver to support devices implementing a simplified version of the USB Integrated Circuit(s) Cards Interface Device (CCID) specification.

The purpose of this assignment is to demonstrate your ability to read and comprehend technical specifications, and apply low level C programming skills to an operating system kernel environment.

You will be provided with the USB CCID specification, a virtual USB device on the COMP3301 virtual machines, and userland code to test your driver functionality.

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. 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 USB CCID protocol provides a standard for a host to communicate with a smartcard reader, which in turn communicates with an actual smartcard. Physically, a smartcard reader can be a standalone device that plugs into a computer, or built into another device like a keyboard. Smartcards are then inserted into the reader so the host can use the functionality they provide. Some devices combine the reader and smartcard functionality into a single integrated device, eg, Yubico Yubikeys or Feitan security keys.

Smartcards provide cryptographic services that can be used for authentication, encryption, and signature generation and verification. The USB CCID specification defines commands and responses that enable communication with and use of these services by the host system. Examples uses of these services include using a smart card for authentication or an extra factor during authentication, as an SSH key, or to cryptographically sign a payload.

Several operating systems, including Windows, macOS, and Illumos, have implemented kernel drivers to support CCID and smart card functionality. On OpenBSD and other UNIX-like operating systems such as Linux there is no kernel support for CCID functionality. Instead, these systems often use the PCSC lite middleware. PCSC lite provides a high level API for working with the service provided by smartcards, but ultimately ends up opening a ugen(4) device entry and communicate with a smartcard using the USB CCID protocol itself.

OpenBSD could benefit from the implementation of a dedicated CCID device driver. Benefits would include:

- Better access control for smart card devices by having them use separate `/dev` entries. ie, if a `ccid(4)` driver claims all USB devices supporting the CCID protocol and only allowed access to their functionality via `/dev/ccid*` device special files, this would enable their access to be managed via file system permissions independently from all the other USB hardware that is supported via `ugen(4)`.
- A dedicated hardware driver could enable concurrent use of the reader by multiple processes by providing transactions and scheduling of commands on the underlying hardware, while `ugen(4)` only supports use of a USB device by a single process.
- Kernel support for the USB CCID protocol could enable use of smartcard cryptographic services by other kernel functionality, eg, as a key to unlock `softraid(4)` crypto devices.

In this assignment you will be developing a kernel driver implementing a subset of the USB CCID specification for OpenBSD. Assignment 3 will be building on top of the Assignment 2 functionality.

2.1 Smartcards and ISO7816

The smart cards supported by CCID readers implement the ISO standard for smartcards, ISO7816. For the most part, in this assignment, you will be treating the actual communication with the smartcard as opaque packets of data, but some high-level details of what they contain may be useful.

ISO7816 specifies 3 layers of communication protocols:

1. The physical interface layer, which specifies electrical signalling etc
2. The transport layer, which deals with Transport Protocol Data Units (TPDUs)
3. The application layer, which deals with Application Protocol Data Units (APDUs)

CCID readers are required by the specification to implement at least the physical interface layer of this model, and most of them also implement (or at least partially implement) the TPDU layer as well. In the CCID class descriptor, readers advertise which layers they implement and which layers the operating system or host software is required to implement instead.

User-level applications that communicate with smart cards generally expect to send and receive APDUs. They send a *command APDU* to the card, and receive a *response APDU* in reply. All communication is initiated by the application, and never by the card itself.

Command APDUs include an *instruction number* (abbreviated `INS`) which specifies the exact operation the application is requesting the card to perform. They also include two 8-bit fixed parameters `P1` and `P2` and can optionally include an additional data field of arbitrary length.

The values of `INS` can be scoped to the particular smart card *application* in use, in addition to some standard values that are always required by ISO7816. For example, `0xA4` is the standard `INS` value for "SELECT", a command which changes the selected application or object for subsequent operations to use. Other `INS` values such as `0x03` might correspond to different operations depending on the context established by the last SELECT command.

Response APDUs consist of an optional arbitrary amount of data, and two bytes known as the "status word". The status word (`SW`) lets the application know whether the command was successful or not.

User-level applications provide command APDUs to the operating system as a string of bytes, and expect to receive response APDUs as a string of bytes too, with the last 2 bytes of the response always being the status word.

To send these APDUs to the card, they have to be encapsulated in a TPDU layer protocol. There are two different TPDU protocols, referred to as `T=0` and `T=1`. Both of these involve making some changes to the layout of the APDU, and adding additional headers or information to make the APDU easily transportable to the card. Depending on the length of the APDU, it may also be split into multiple TPDUs.

If the CCID reader does not implement TPDU support, then the operating system or host software will have to deal with this encapsulation process. Otherwise, if the CCID reader advertises "APDU-level" support, then the OS only needs to send an APDU as-is to the reader, and it will handle TPDUs automatically.

In this assignment you will only be required to support readers which advertise APDU-level operation – dealing directly with TPDUs is not required.

You can access the full CCID class specification at: https://stluc.manta.uqcloud.net/comp3301/public/usb-ccid-r110.pdf

3 Specifications

You will implement a driver called `ccid(4)` that will attach to USB CCID readers, and provide access to that reader and an inserted smart card via a character device special file under `/dev`.

You may assume that userland tools which use your driver will directly open the device and use the `read`/`write`/`ioctl` interface specified below. You do not need to implement any additional userland components to go with it.

3.1 USB Functionality

Your `ccid(4)` driver will attach to all USB interfaces that advertise the Smart Card Device Class, subclass 0, and protocol 0, as per the USB CCID specification.

The attach handler for the hardware driver should enumerate all the USB endpoint descriptors to find the Bulk in and out endpoints for communicating with the reader, and fetch the Smart Card Class Descriptor.

The driver does not have to implement support for detaching the driver if the CCID reader is unplugged, and hotplug will not be tested as part of this assignment.

The driver also does not need to implement handling of an interrupt endpoint or provide event notification or insertion or removal of a smartcard.

3.2 CCID Functionality

Since the USB CCID specification is a large document including a lot of features for different types of smart cards and readers, in this assignment you will only be required to implement a subset of the full spec.

In particular:

- **Single-slot only**: you may assume that your reader will only advertise a single "slot" and can only ever contain one card. You do not have to examine or process the `bMaxSlotIndex` in the class descriptor, and the `bSlot` field in outgoing CCID messages should always be set to `0`.
- **APDU-layer only**: you are only required to support readers which advertise APDU-layer operation (meaning that your driver does not have to implement `T=0` or `T=1` TPDU layer encapsulation)
- **Automatic clock, voltage and baud rate only**: you are not required to work out what voltage, clock speed or baud rate is required by a card, and should only support readers that handle this automatically.
- **PPS support not required**: along with being APDU-layer only, and only supporting automatic clock speed operation, you are not required to implement PPS

- **Timeouts not required**: you are not required to implement command time-outs, deal with calculating the "correct" wait times for commands or to process command time extension requests (`RDR_to_PC_DataBlock` with `bStatus` = `0x80`)
- **Aborts not required**: you are not required to implement support for the `ABORT` control request or send `PC_to_RDR_Abort` CCID messages, or deal with any of the semantics related to these.
- **No automatic power-on**: you may assume the CCID reader requires an explicit power-on CCID message to power up the ICC (i.e. the `Automatic activation of ICC on inserting` CCID feature is not available)

You are only required to implement enough of the CCID commands to provide the device interface in this specification. There are CCID commands (e.g. `PC_to_RDR_Mechanical` for readers which can physically eject or lock their card slots) which you do not need to implement any support for, since they have no matching `ioctl` or requirement in the device interface.

Part of the objective of this assignment is reading and interpreting the CCID specification, and working out the exact commands which will be required to implement the required behaviour. You should pay close attention to the descriptions of each command in the specification, and read the example exchanges towards the end of the document, to work out which are required.

3.3 Device Special File interface

Because `ccid(4)` only supports readers with a single smartcard slot, there will be a one to one relationship between the CCID readers and smartcards.

The `ccid(4)` device special file should use major number 101 on the amd64 architecture for a character device special file. The `ccid(4)` device minor maps to the `ccid(4)` unit number, which in turn maps to the one smartcard that will be present in that CCID reader in slot 0.

The driver does not have to support non-blocking operation.

3.3.1 open() entry point

The open() handler should ensure that the minor number has an associated kernel driver instance with the same unit number attached. Access to the device special file should be exclusive, meaning every successful open should have a close before further open calls would be allowed.

Access control is provided by permissions on the filesystem.

The open() handler should return the following errors for the reasons listed. On success it should return 0.

ENXIO no hardware driver is attached with the relevant unit number
EBUSY the device special file is already open
ENOMEM the kernel was unable to allocate necessary resources

3.3.2 write() entry point

The write handler allows userland to send an APDU to a smart card that is inserted into the reader.

An entire APDU must be provided to `write()` in a single call – one APDU cannot be spread across multiple `write()` calls. A call to `writev()` is treated as a single write (even if it uses multiple `iovecs`).

The `write()` handler encapsulates the APDU bytes from userland in a CCID message and submits it to the bulk-out message endpoint.

The write must block until at least the command submission to the CCID reader has been completed, and *may* block until a response is received if you wish (this might make your code simpler, depending on how you approach it).

EBUSY a request is already in progress – either waiting for a response from the reader or a response has been received but `read()` has not yet consumed it

EACCES the device is open read-only

3.3.3 read() entry point

read() gets a response from the smart card inserted into slot 0 of the CCID reader.

The `read()` handler will wait for a response CCID message from the reader related to the most recent `write()` call that submitted a request. It should block until the response is available.

The response (or the portion of it that will fit) will be transferred to the buffer supplied by userland.

EACCES the device is open read-only

EIO the CCID reader returned an error response to the CCID-level command (e.g. no card/ICC present, hardware error). Note that this does *not* include errors returned as an APDU-level error by the card (e.g. a `SW` other than `0x9000`), which are processed as normal response APDUs.

ENOMSG no write request has been made

3.3.4 close()

releases any resources that were allocated by open, and makes the device available for another open.

If a request is currently in-flight (i.e. a command or USB transfer is currently awaiting a response), then `close()` should block until the request has completed.

Any outstanding responses from the card which are waiting for a `read()` are discarded when `close()` is called, and the card will then be powered down (with the same operation as used for `CCIDIOC_POWEROFF`).

close must not return an error.

3.3.5 ioctl()

`ccid(4)` specific ioctls used by the userland testing tool are provided in `src/sys/dev/usb/ccidvar.h`. This header specifies both the `ioctl` magic numbers (`CCIDIOC_*` macros) and the structs used as arguments to them.

You must not modify the `ioctl` definitions provided: they are the interface that is required for testing.

3.3.5.1 USB_DEVICEINFO

`ccid(4)` should implement the USB_DEVICEINFO ioctl to provide information about the USB device the CCID interface is present under. Unlike `USB_DEVICEINFO` requests against `usb(4)` devices, userland does not have to specify an address in the `udi_addr` member of the `usb_device_info` struct because the `ccid(4)` driver should already maintain a link to the relevant USB device.

3.3.5.2 CCIDIOC_GET_DRIVER_INFO

The `CCIDIOC_GET_DRIVER_INFO` ioctl should provide information about the hardware device driver to userland. The information includes the name of the driver, the name of the driver instance created by that driver, and the unit number of the driver instance.

3.3.5.3 CCIDIOC_GET_LAST_ERR

The `CCIDIOC_GET_LAST_ERR` ioctl should provide more detailed information about an error which resulted in `EIO` being returned by `read()` or another `ioctl` (such as `CCIDIOC_POWERON`).

It contains both a numeric error code and a string description which the driver should fill out based on the error description tables in the CCID specification. The string should be of the format `ERROR NAME: Cause string`.

For example, the numeric error code `-2` should return a string similar to `ICC_MUTE: No ICC present, or CCID timed out while talking to the ICC`.

Our tests will look for the "ERROR NAME" at the start of the string to match the CCID specification, but will not require any particular cause description following it.

The "last error" state is reset each time the device is `open()`ed. If no error has occurred, the ioctl should return `EIO` instead of a `ccid_err_info` struct.

This ioctl may fail with errors for the following reasons:

EIO no such error has occurred since the device was last `open()`ed

3.3.5.4 CCIDIOC_GET_DESCRIPTOR

The `CCIDIOC_GET_DESCRIPTOR` ioctl should provide a copy of the Smart Card Class Descriptor to userland, represented by `struct usb_smartcard_descriptor` in `src/sys/dev/usb/usb.h`.

3.3.5.5 CCIDIOC_POWERON

This ioctl sends a poweron message to the reader and waits for the associated response.

If the ICC is already powered up, this operation should return success.

This ioctl may fail with errors for the following reasons:

EACCES the device is open read-only
ENOMEM the kernel was unable to allocate necessary resources
EIO the reader returned an error response (e.g. no ICC/card present, parity error, etc)

3.3.5.6 CCIDIOC_POWEROFF

This ioctl sends a powerdown message to the reader and waits for the associated response.

If there is an outstanding operation due to `read()` or `write()` when this ioctl is entered (e.g. a USB transfer is in progress, or the driver is waiting for the reader to respond to a command), it should block until that operation completes before sending the power down message.

Any outstanding response from the card which is currently waiting for a `read()` call to be made will be kept and not discarded by `CCIDIOC_POWEROFF` (unlike a call to `close()`).

This ioctl may fail with errors for the following reasons:

EACCES the device is open read-only

ENOMEM the kernel was unable to allocate necessary resources

3.3.6 poll/kqueue

`ccid(4)` does not need to support non-blocking operation for this assignment.

3.4 Testing

3.4.1 Userland tool

In the provided code (see section below for instructions, similar to A1) we have given you a userland tool named `ccidctl`, which you may use to test your driver.

You can build `ccidctl` by running `make` in the directory `usr.sbin/ccidctl` as per usual for a userland tool.

The commandline interface for `ccidctl` is summarised below:

```
usage: ccidctl ls [-v]
       ccidctl show [-v] ccid0
       ccidctl reset ccid0
       ccidctl ykselect ccid0
```

3.4.1.1 ccidctl ls

`ccidctl ls` iterates over `/dev/ccid0`, `/dev/ccid1`, `/dev/ccid2` and `/dev/ccid3` and shows information about each device. The amount of information can be increased with the `-v` flag

3.4.1.2 ccidctl show

`ccidctl show` opens the specified device special file and shows information about the CCID device. The amount of information can be increased with the `-v` flag

3.4.1.3 ccidctl reset

`ccidctl reset` opens the specified device special file and issues power off and power on ioctls to reset the device.

3.4.1.4 ccidctl ykselect

`ccidctl ykselect` issues a command to the smartcard requesting the selection of the Yubico OTP applet. On success, it will output the YubiKey firmware version.

The simulated CCID device in your course-provided virtual machine implements this applet and command, and will return firmware version `33.0.1`.

3.4.2 Other testing

You may wish to alter `ccidctl` during your testing, and we strongly recommend that you do. Make sure to test invalid sizes and arguments to `ioctl` commands and `read` and `write`, performing operations out of order, and other similar error conditions.

We will not be using your `ccidctl` code in your repository for marking, so you are free to modify it as you wish and keep those modifications in your `a2` branch.

Note that the `ioctl` interface of the device must be kept exactly as it is given to you (i.e. the `ioctl` magic numbers and their arguments and structs defined in `dev/usb/ccidvar.h`).

We will be testing for error conditions (not just success) when marking your submission for this assignment.

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 `make obj; make config; make` in `src/sys/arch/amd64/compile/GENERIC.MP`. relative to your repository root.

3.7 Provided code

A patch will be provided that includes source for the `ccidctl` userland program, and the `src/sys/dev/usb/ccidvar.h` header file containing the ioctls that `ccidctl` expects the `ccid(4)` driver to implement.

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/a2-base.patch

You should create a new `a2` 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 a2 openbsd-7.1
$ ftp https://stluc.manta.uqcloud.net/comp3301/public/2022/a2-base.patch
$ git am < a2-base.patch
$ git push origin a2
```

3.8 Recommendations

You should refer to the course practical exercises (especially pracs 4 and 5) for the basics of creating a USB device driver and attaching to a device.

The following are examples of some APIs you will likely need to use in this assignment:

- `usbd_open_pipe(9)`
- `usbd_transfer(9)`
- `uiomove(9)`
- `tsleep(9)`

You may also wish to refer to the code of some existing device drivers in the kernel, in particular:

- uvideo(4)
- cdce(4)

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 `a2` 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 `a2` branch should consist of:

- The openbsd-7.1 base commit
- The a2 base patch commit
- Commit(s) implementing the `ccid(4)` driver

We recommend making regular commits and pushing them as you progress through your implementation. As this assignment involves a kernel driver, it is quite likely you will cause a kernel panic at some point, which may lead to filesystem corruption. Best practise is to commit and push before every reboot of your VM, just in case.

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 the Practical exercises.

Tests will be run against the same simulated CCID device available in your personal virtual machine for the course.