6841 Something Awesome - Fiduido (FIDO Authenticator on an Arduino)

Hamish Cox, COMP6841 22T1

1. Introduction

The project I chose was to DIY a USB authenticator, specifically a FIDO compliant one - like Yubikeys or Google Titan keys.

These keys act as another authentication factor available to the user, and work by storing private keys on a read-or-reset-only chip, inaccessible to any connected computer except via a specific protocol to request that authenticator sign challenges to be used to authenticate on websites and applications.

My original goal/minimum viable product was to implement a working USB authenticator, however my research and effort over the term slowly showed this was infeasible. Unfortunately, this realisation occurred rather late into the project, so I decided to make my goal easier to achieve rather than swap project entirely. My authenticator would need to be recognised by the computer and be able to complete the handshake/initialisation process so that the computer can learn about it. Even then, a key technical challenge was still too complex to overcome in time.

I considered creating a Python interface to simulate the HID connection to the authenticator and allow for some level of demonstration, but the time I have left has made this was also too much to hope for. Instead, I've submitted my unreliable and incomplete CTAPHID implementation alongside this report.

2. Table of Contents

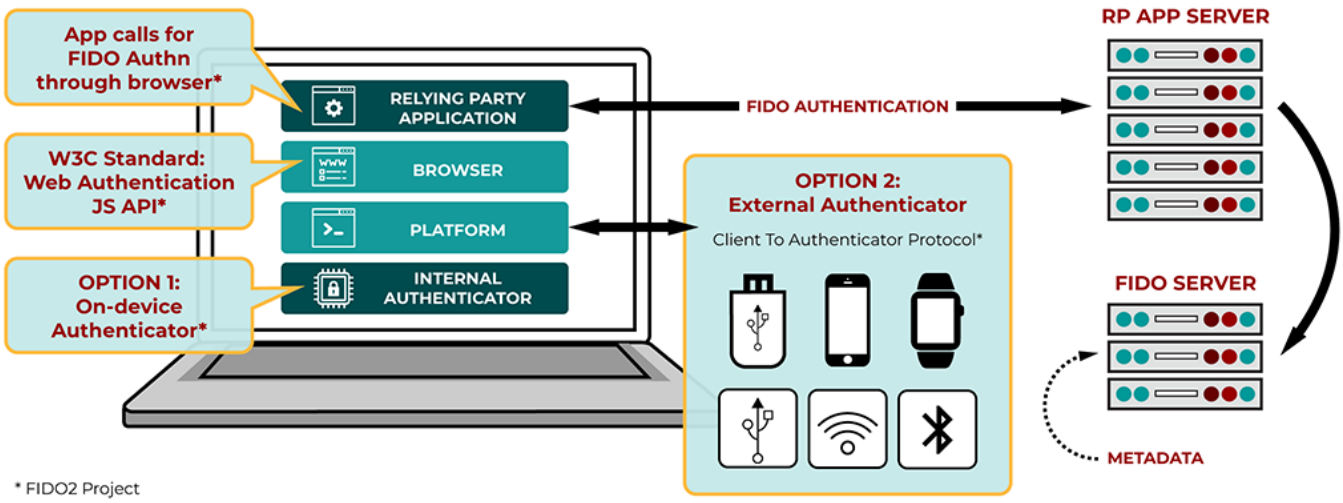
You can use this to skip to more interesting parts, such as the technical challenges I've mentioned above, or my reflections on the project and authenticators/authentication overall.

3. Understanding FIDO2 and CTAP

3.1. FIDO2 Overview and Terminology

If you can't be bothered reading this, skip to the Summary section.

This is a graphic from the FIDO Alliance's website outlining how their FIDO2 specifications interact.



In particular, we're interested in the external authenticators. These are communicated with by the platform (OS) via the Client To Authenticator Protocol (CTAP), which we'll get to in more detail later. First though, here are an overview of the other components and some other terms you may see thrown around:

FIDO and FIDO2

"FIDO" is not a technical term, rather a marketing one, and stands for **F**ast **ID**entity **O**nline, and refers to the set of specifications written by the FIDO Alliance, "an open industry association ... to help reduce the world's over-reliance on passwords". An authenticator is FIDO compliant if it correctly implements the relevant FIDO specifications and can become FIDO Certified by being certified by the FIDO Alliance. FIDO2 specifications are the FIDO Alliance's second set of specfications.

UAF and U2F

UAF (Universal Authentication Framework) and U2F (Universal 2nd Factor) are legacy frameworks/protocols that have been deprecated in favour of FIDO2. Note that U2F is also known as CTAP1, since CTAP2 devices can be backwards compatible with U2F applications.

WebAuthn

WebAuthn (Web Authentication and Authorisation) is the JavaScript API used by websites to communicate with authenticators, with the platform/OS acting as proxy/translator. This is a joint W3C and FIDO Alliance specification and is part of FIDO2.

CTAP

CTAP (Client To Authenticator Protocol) is the protocol for interaction between the platform/OS and an external authenticator (a FIDO security key such as a Yubikey or Google Titan Key). Note that CTAP is compatible with multiple forms of communication: USB, NFC and BLE (Bluetooth Low Energy).

CTAP2 is the FIDO2 version of CTAP, and can be backwards compatible with U2F/CTAP1.

Summary

To summarise:

- **FIDO:** Term/name referring to specifications written by the FIDO Alliance, a group of companies who want to reduce the risk posed by passwords. FIDO2 is the second set of specifications.
- **WebAuthn:** FIDO2 JS API for interacting with authenticators via the platform/OS.
- **UAF and U2F:** Legacy frameworks/protocols, deprecated in favour of FIDO2.
- **CTAP:** Protocol for interaction between platform/OS and external authenticators. Compatible with USB, NFC and BLE.

3.2. CTAP

Now that we understand the basics, we can dive into the spec relevant to creating an authenticator: CTAP.

At the time of writing, this specification can be found here.

There are three parts or layers to this protocol - the authenticator API, how messages are encoded, and how these messages are passed between the platform and authenticator.

Authenticator API

The authenticator API layer of CTAP is the series of commands that actually request cryptographic operations from the authenticator, in addition to configuration of PINs, fingerprints and other features. For some commands, algorithms are provided to make the correct operation of the authenticator clear.

For example, the authenticatorGetAssertion command defines how the authenticator should sign a challenge provided by the application.

The specification also outlines error codes and parameter structures for each command.

Message Encoding

These commands (specifically, the parameters, which vary a lot between commands) are communicated using CBOR (Compact Binary Object Representation). CBOR is effectively a binary form of JSON. Due to the usually quite limited resources on authenticators, a particular form of CBOR is used, called the **CTAP2 canonical CBOR encoding form**.

Sidenote: This is just a really cool encoding - why aren't we using this in REST APIs and GraphQL and whatnot?

Transport-Specific Bindings

And now we reach actual communication. But this can't always be done the same way! USB authenticators get plugged in, but NFC and BLE devices are wireless, and each has different protocols and interfaces themselves.

And so, since I'm focusing on USB we must look at...

3.3. CTAPHID

This makes more sense as **CTAP *over* HID**. This section of the CTAP specification defines how messages (consisting of a command and its' parameters) should be transmitted between the client and authenticator.

CTAPHID has a number of features that it was designed with:

- Driver-less installation on all major platforms (see HID for how).
- Support for multiple applications, accessing the authenticator concurrently.
- CTAPHID device discovery (again, see HID for how).
- Fixed latency response and low protocol overhead.

The CTAPHID specification defines transactions that consist of a request (either a authenticator API command, as explained above, or a CTAPHID command) and a response (the expected response or an error).

These requests and responses are transmitted as single messages that, if necessary, are transmitted in packets. Packets are a fixed size, defined by the authenticator, but are limited to 64 bytes or less, due to the restrictions of HID.

Every transaction must be completed before another can be initiated. A response cannot be sent unless a corresponding request was received (with the exception of keep alive messages, used by U2F/CTAP1).

To achieve support for multiple applications using the same authenticator, CTAPHID uses logical channels that are unique for each application. Whenever an application wants to begin using an authenticator, it sends an initialisation request that is responded to with a channel ID that should be used in subsequent requests.

HID

CTAPHID works over the HID (Human Interface Device) protocol, which defines how common USB devices used for user input such as mice, keyboards and game controllers can communicate with the computer without the installation of drivers. This enables CTAPHID authenticators to be connected to a computer and immediately begin to be used.

HID transmits reports (CTAPHID packets) of a fixed size chosen by the device. The size of reports/packets is defined by the HID report descriptor, a special type of USB descriptor (see USB) that describes HID capabilities, alongside the HID device's usage page (device category) and usage (place in that category). CTAPHID devices use the FIDO Alliance usage page (`0xF1D0`, good isn't it?) and the CTAP device usage (`0x01`). This allows the OS to detect CTAPHID devices easily by checking for this particular usage page and usage.

USB

USB is an extremely complicated protocol - so here are the relevant parts:

- Data is sent to/from device endpoints - USB devices can define a couple (depending on hardware), and assign each a direction (IN to the USB host/computer, or OUT to the device).
- These endpoints are defined by interfaces, which are contained within configurations, which are all described by descriptors.

At the USB layer, CTAPHID requires that packets are sent and received on two separate USB endpoints.

Did you need to go through all this?

You might have wondered if I actually needed to outline how USB and HID work. Well given how I was programming a microcontroller, I needed at least a basic understanding of how USB and HID work, so I may as well write at least a bit about it. There's a whole lot more I've learnt about both that I haven't put into writing.

Commands

Now that we know how CTAPHID transmits messages, we can look at what those messages are. In addition to the authenticator API, CTAPHID defines some further commands related to interacting with USB authenticators, mostly relating to handling errors and manging communication at the CTAPHID layer, but also a 'wink' command that tells the authenticator to attract the user's attention, such as through a flashing light or a sound.

4. Building an Authenticator

From the CTAP(HID) specification, we can see that an authenticator is simply a USB device that implements CTAPHID. So how do I build a USB device?

4.1. Hardware

Arduinos are ready to use microcontrollers attached to a PCB for power, general purpose IO pins, and other useful features. They're designed to be easy to get started with and are the goto microcontrollers for most beginner embedded system projects. They are also readily available rather than waiting a month for shipping from overseas.

However, most Arduinos (e.g. the most common: the Arduino Uno) do not support regular USB communication, rather the onboard USB port only support serial and the chip programming interface. Instead, the Arduino Leonardo is the board that supports the creation of USB devices. It has 6 USB endpoints, and CTAPHID only requires 3 (IN, OUT and the control endpoint)! It was perfect for this project, and I went and picked up one from Jaycar.

4.2. Software Libraries

And now I had reached the bulk of the work. Now I had gone through the CTAPHID spec and I had realised how much work getting a full implementation would be. I decided to scale the project down to having an authenticator able to provide it's capabilities to the computer. Yubico (the creators of the popular Yubikey devices) provides a command line tool to interact with FIDO2 authenticators. This is (roughly) what I wanted

my authenticator to be able to return when the command was run (this is the output returned by my own Yubikey, the comments after // are added by me):

```
❯ fido2-token -I ioreg://4295261606
proto: 0x02 // CTAP version.
major: 0x05 // Device version.
minor: 0x04
build: 0x03
caps: 0x05 (wink, cbor, msg) // CTAPHID capabilties: supports WINK, CTAP2
commands and CTAP1 commands)
version strings: U2F_V2, FIDO_2_0, FIDO_2_1_PRE // Compliant to FIDO2.0,
FIDO2.1 and U2F V2 specs.
extension strings: credProtect, hmac-secret // Supports these CTAP
extensions.
transport strings: nfc, usb // Supports NFC and USB communication.
algorithms: es256 (public-key), eddsa (public-key) // Available
algorithms.
aaguid: 2fc0579f811347eab116bb5a8db9202a
options: rk, up, noplat, clientPin, credentialMgmtPreview // Supports
resident keys, user presence, setting a PIN and the preview credential
management API. It is also not a platform authenticator.
maxmsgsiz: 1200 // Maximum message size.
maxcredcntlst: 8 // Credential limits.
maxcredlen: 128
fwversion: 0x50403
pin protocols: 2, 1
pin retries: 3
uv retries: undefined // No user verification (e.g. biometrics).
```

Arduino IDE and Libraries

While this would be a much more achievable goal, this still assumes that I can successfully implement HID communication on my Arduino. To get started with this, I looked at the Arduino USB libraries. PluggableUSB in particular. The issue was that the documentation and API for these was extremely limited, and didn't support what I need (at least, I thought so - looking at the API again now that I understand USB a lot more, it may have what I need, I'll have to investigate it another time).

The Arduino IDE is also really bad - zero autocomplete and cryptic error messages. Also, since I was communicating over USB and the Arduino IDE needs to program over USB, the IDE struggled to upload my compiled code. There is a way around this but it wasn't consistent when used with the Arduino IDE.

I tried using the Arduino extension in VS Code, but the mess of preprocessing that the Arduino toolchain uses got in the way.

LUFA

Well if Arduino's standard library isn't going to work I'll have to take a look at my other options. One library in particular seems to be the go-to for USB on AVR microcontollers (the Arduino Leonardo has an ATMEGA32U4, which is an AVR microcontroller). This library is LUFA (Lightweight USB Framework for

AVRs), which provides low-level control of the chip's USB functionality. It also provides demos for various common use cases. However, it doesn't work within Arduino's ecosystem.

The AVR Toolchain

Now that I'm working with plain C code, I need to compile and flash my code to the microcontroller, rather than rely on Arduino's version. To do this, I installed avr-gcc and avrdude, which is the AVR compiler from the AVR standard library implementation and a program to flash the code to the chip's program memory.

There's some particular Arduino Leonardo setup to be done, but thankfully the author of LUFA provides a configuration for it.

4.3. Writing Application Code (and it's challenges)

Once I had LUFA's demos running, I began writing code to implement CTAPHID. Unfortunately, this was a lot harder than it sounds.

Low Level vs Class Drivers

Once I had LUFA's demos running, I began writing code to implement CTAPHID. Unfortunately, this was a lot harder than it seems. LUFA provides two different demos for each application - one that uses LUFA's higher level USB/HID class driver implementations (functions and callbacks to implement common uses quickly), and one that does the same but by implementing that functionality itself.

Due to a 'technical limitation' (that doesn't get mentioned anywhere except in a random forum post that gets moved to private emails...), the LUFA class drivers for generic HID devices doesn't support having an OUT endpoint, which is required by the CTAPHID spec. I suppose I'll have to use the low level version - except I barely understand how USB works...

USB and HID

I get started adapting the low level generic HID device demo to work for my CTAPHID device. Soon enough I have my Arduino showing up as "COMP6841 Something Awesome Project" in my computer's USB device list, and being detected as an authenticator by Yubico's CLI tool:

```
❯ fido2-token -L
ioreg://4295262143: vendor=0x1050, product=0x0407 (Yubico YubiKey
OTP+FIDO+CCID)
ioreg://4295262114: vendor=0x2786, product=0x6837 (Hamish Cox COMP6841
Something Awesome Project)
```

But when I actually try to send and receive messages, I get extremely unhelpful errors from my host-side application (the demos provide Python scripts to send and receive data). I began debugging *everything* my code is doing - I dig into LUFA's internals, I dig into the USB spec, I look for debugging tools.

At some point, a good 3 hours into this issue, in my 3rd check of the differences between the working demo and my code, I notice the endpoint addresses: I'm using 0x81 and 0x1, they use 0x81 and 0x2. The 0x8 part indicates direction. Sure enough, by changing an endpoint address, it begins to work as expected.

But why had I changed it to begin with? Well, the CTAPHID spec is written for people who are familiar with USB and HID to begin with... so the example endpoint addresses they provided, that I was using, were invalid.

This was just one of my issues with USB communication, but definitely the most frustrating.

Memory

Even getting past those issues (and they aren't solved - communication is frustratingly inconsistent), actually implementing the logic of a FIDO2 authenticator would be extremely difficult: the ATMEGA32U4 on the Arduino Leonardo has 2.5KB of RAM. Two point five. **Kilobytes**. Not sure how I'd be doing cryptography with that much, since I didn't get that far, but it should be possible, considering the size of Yubikeys (like the Yubikey 5C Nano, which is absolutely tiny).

4.4. What I've Written

So what have I actually written? You can find a GitHub repository with my code **here**.

`FidoHID.c` contains `main` and USB communication related functions, including handlers for the CTAPHID commands that I did implement.

`Descriptors.c` (and `.h`) contains definitions for the various USB descriptors, and a callback to handle responding to USB control requests for those descriptors.

`ctap2hid_message.c` (and `.h`) contains the message struct and functions for reading and writing CTAPHID messages to/from packets.

`ctap2hid_packet.c` (and `.h`) contains the packet struct and functions for checking packet properties.

`packet_queue.c` (and `.h`) contains a queue implementation (without malloc, because 2.5KB of RAM) to queue up packets for processing (when read from the host) and writing (when they should be sent to the host).

The `Config` directory holds some config `.h` files. `HostTestApp` contains a modified version of the demo's Python script for testing. `GenericHID` is the demo code I was working off of.

5. Reflection and Analysis

5.1. On My Project

My project had some major flaws in its' execution that prevented me from completing it to the standard I was hoping, but I also learnt a lot from the experience, including learning the technical aspects of FIDO2 and learning how I can more effectively manage my time and procrastination.

Unfortunately, the 30+ hours spent on this project are mostly in the form of writing and debugging code, which is hard to convert to time spent after the fact - since I didn't realise how much my implementation issues would impact my project, I didn't spend the effort I should have documenting them (issue, time spent, fix, etc.).

Another key flaw in my project's execution was leaving the technical aspect to the later half. I spent the first half doing more basic research which limited my ability to tell how large a project this was. If I had

determined more quickly what I needed to be doing, I could have begun encountering and solving my technical issues sooner.

This would also have helped solve another flaw: procrastination. Since I enjoy solving technical problems, this would have been a more effective strategy to ensure I was more on top of this project. I know I prefer technical work, but I haven't worked on many self-guided/directed projects that are a combination of technical skills and writing, so hadn't considered how I could effectively make myself get started. This is an important lesson for me that I can apply to later projects.

I also learnt a great deal about the authentication process by diving into the details of even a small section of the whole FIDO2 system. This has allowed me to understand the underlying technology in my personal FIDO2 devices, and know which authentication challenges it solves and doesn't solve.

5.2. On Authenticators

I want to preface this section with:

- I can't be sure of many of my conclusions in this section due to my simple understanding of HID and USB, so parts of this may be inaccurate, but rather than get hung up on that, I'm analysing based on my current understanding.
- I can't really analyse the parts of FIDO2 I didn't focus on. In particular, I unfortunately don't know enough about the cryptographic side of FIDO2 to analyse it. Instead I'll be focusing on the transport layer and the issues with the authenticator's communication with the host platform.

An important thing to note about HID devices is that they are generally available to every process on the system. It's hardly a security risk to read data from a joystick or a barcode scanner (actually maybe this could be, but you get the point). Any process can generally read and write to any USB (or in this case, HID) device. And so we could just listen to the communication between the authenticator and the PC.

If listening to the output the authenticator is sending to other processes isn't doable, then why not make our own connection to it? Initialise enough channels and you can probably figure out some already configured channel ID and begin sending requests that rely on that channel's state. This could be mitigated by generating channel IDs using a cryptographic random number generator. But doing this on a 2.5KB RAM chip wouldn't be great for performance.

Of course, do you really need access to another channel? If you can communicate with the authenticator, you can just create your own channel and start asking this user to verify things. Suppose you send a request for logging in to a server, and it responds with the FIDO2 details it expects to your process. Time or fake the prompts well enough and you could possibly get verification for this service when the user thinks they are verifying for another.

Even then, what if we can just deny the user service by sending a ton of requests and tanking it's performance? Or send 10 invalid PINs, which locks the authenticator and requires the user to reset it?

Bottom line is that any access to authenticators is bad news. But any protection for HID devices (such as for keyboards, to prevent really easy keylogging) is OS specific and not defined by the FIDO2 specifications. In fact, on my Macbook, I can interact with my Yubikey just fine with an open source command line program (written by Yubico, in fact) that doesn't even require root!

So I now know that my authenticator really *mustn't* be plugged into untrusted computers, contrary to Yubico's advice.

6. Continuing This After 6841

After I've finished this project and have some time, I would certainly like to come back to this code and sort out the technical issues, and hopefully get a working authenticator. I also would like to try creating some of the attacks I mentioned in my analysis of authenticators.

I really need some time away from this project though, since this last week of trying (and failing) to get something working has really stressed me out.

7. Appendices

7.1. Appendix A: GitHub Repository Link

My code (and this report in markdown format) can be found here: https://github.com/HamishWHC/FidoHID

7.2. Appexdix B: CTAP Specfication Link

This is the version of the CTAP spec that was accurate at the time of writing, and the one I followed: https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-20210615.html

7.3. Appexdix C: Blog Post Links

I had very few, sorry, but they can be found here:

- My initial brainstorming post (and setting insane endgoals): https://www.openlearning.com/u/hamishcox/blog/W1SomethingAwesomeTopicBrainstorming/
- An info dump after I begun the technical side and realised how much work it was: https://www.openlearning.com/u/hamishcox/blog/W6SomethingAwesomeInfoDump/