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.

Rather than submit my broken implementation (although it is available), I've written a Python program which allows the user to interact with my authenticator's firmware, running on the computer instead of an Arduino.

2. Table of Contents

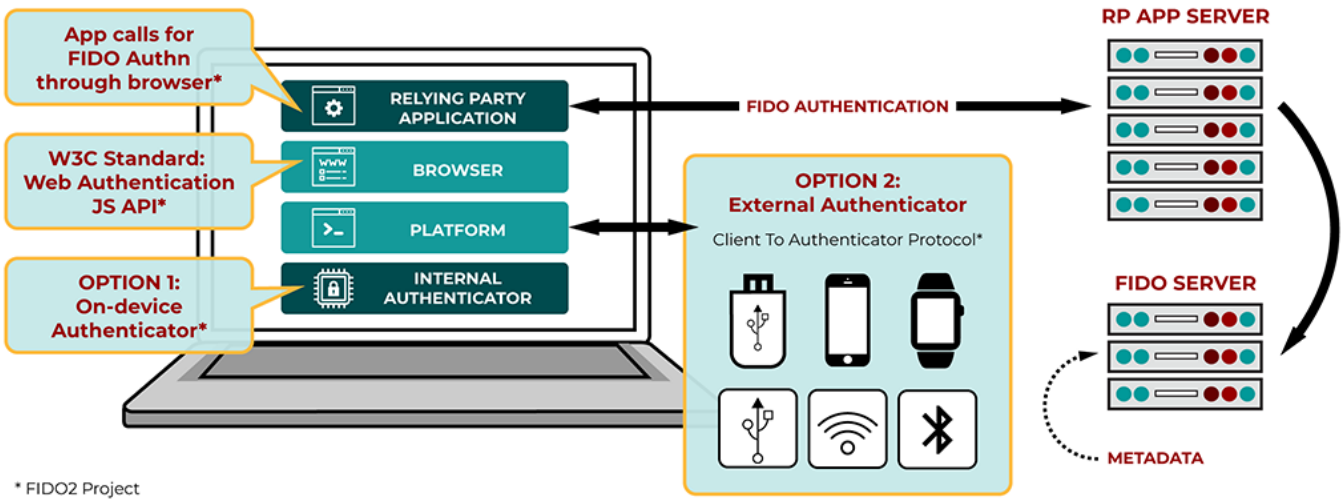
You can use this to skip to more interesting parts, such as the technical challenges I've mentioned above, or my reflections on the project and authenticators/authentication overall.

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.

4. Plan A: Building a Physical Authenticator

CTAP commands, CTAPHID protocol.

4.1. Hardware

Leonardo because cheap + ready to go + supports what I need.

4.2. Software

Going through the process of writing the firmware for the Arduino - or more specifically the ATMEGA32U4.

Arduino IDE and Libraries

(and why I dislike them)

The AVR Toolchain

2.5KB isn't much to work with. Also avrdude and avr-gcc, etc.

LUFA

(and why C is better than ArduinoC)

4.3. Challenges of HID and USB

5. Plan B: Simulating an Authenticator

5.1. Why It Was Necessary