



**FACULTY
OF ELECTRICAL
ENGINEERING
CTU IN PRAGUE**

FIDO2 USB Security Key

B4MSVP Final Report

Martin Endler

Open Informatics – Cybersecurity

June 2024

Supervisor: Ing. Jan Sobotka, Ph.D.

Abstract

FIDO2 is a set of standards based on asymmetric cryptography that enables easy, secure, and phishing-resistant authentication. The strong support from Google, Microsoft, Apple, and other major technology companies in the FIDO Alliance is driving the adoption of FIDO2 across the industry.

This project focuses on creating a new open-source FIDO2 USB hardware external authenticator. We provide an overview of the Web Authentication and CTAP2 standards, which are together known as FIDO2. We also review existing projects related to our goal. Then, we proceed to the actual implementation, which represents the main contribution of this project. We document our decisions and the most important parts of the implementation. All code for this project is publicly available on GitHub.

We hope our work will contribute to the popularization and better understanding of the FIDO2 technology.

Keywords: FIDO2, WebAuthn, CTAP2, USB, security key, passkeys, authenticator, passwordless authentication, multi-factor authentication, MFA, second-factor authentication, 2FA, asymmetric cryptography, public key cryptography

Version: 2024-06-11 11:55 CEST

Contents

1 Introduction	4
1.1 Structure	5
1.2 Project Goal	5
2 FIDO2	6
2.1 Functional Description	7
2.2 Key Benefits	8
2.3 Terminology	9
2.3.1 Relying party (RP)	9
2.3.2 Client	9
2.3.3 Client Device	9
2.3.4 Client Platform	9
2.3.5 Authenticator	9
2.3.6 Credential Storage Modality	10
2.4 Ceremonies	11
2.4.1 Registration	11
2.4.2 Authentication	13
2.5 Additional Resources	14
2.6 Security Key Products	14
3 Existing Work	15
3.1 Open-Source Software and Hardware	15
SoloKeys	15
OpenSK by Google	15
Nitrokey	15
3.2 Other Relevant Projects	16
OpenTitan	16
TROPIC01 (TRuly OPen IC)	16
4 Implementation	17
4.1 Architecture	18
4.1.1 CTAP2	18
4.1.2 CTAPHID	18
4.1.3 USB	19
4.1.4 Authenticator State	20
4.1.4.1 Persistent Storage	20
4.1.4.2 True Random Number Generator	22
4.2 Data Flow	23
4.3 Debugging Interface via UART	24
5 Conclusion	26
5.1 Future Work	26
References	27

Chapter 1

Introduction

Passwords as a means of authentication suffer from many problems. More than 80% of confirmed breaches are related to **stolen, weak, or reused passwords** [5]. Password-based credentials are the target of **phishing attacks**, which are becoming more sophisticated every day. This poses a major threat since 84% of users reuse the same passwords across multiple sites [6]. To limit this threat and to increase the overall security of authentication, *two-factor (2FA)* or *multi-factor (MFA)* authentication flows are being increasingly used [6].

However, the standard MFA mechanisms, including one-time codes delivered via insecure channels (such as SMS, voice call, or email), TOTP (e.g., Google Authenticator), and proprietary push notification-based systems, do not provide sufficient security. Not only are they all still susceptible to phishing attacks, but they also greatly hinder the user experience.

To solve this problem, the FIDO Alliance was launched in 2013 [8]. It develops and promotes strong authentication standards that “help reduce the world’s over-reliance on passwords” [7].

Its latest set of standards, jointly developed with the W3C (World Wide Web Consortium), is called **FIDO2**. It is based on *asymmetric cryptography*, and it enables **easy, secure, and phishing-resistant authentication** for online services (primarily on the web, but it can be used in native applications as well). It supports *passwordless*, *second-factor*, and *multi-factor* user experiences with *platform authenticators* (such as Apple ID with Face ID or Touch ID, Windows Hello, and Google Password Manager on Android) or *external (roaming) authenticators* (such as **FIDO2 security keys**). All major OSs, browsers, and a growing number of websites and applications support FIDO2 [13, 14, 12].

Among the FIDO Alliance’s 250 members are all the major technological companies [10]. Google (one of the FIDO founding members), Microsoft, and Apple have become vocal advocates for passwordless authentication based on passkeys (the end-user-centric term for FIDO2 credentials) since 2022 when they announced their public commitment to expand support of FIDO2 [11].

With the ever-increasing adoption of FIDO2 across the industry [6], it is useful to understand how this technology works and what its benefits are. In order to do that, we decided to **create a new open-source implementation of an external (roaming) FIDO2 authenticator** from scratch.

1.1 Structure

The work is structured as follows:

First, in [Section 1.2](#), we define **the goal** of the project.

Second, in [Chapter 2](#), we establish the necessary theoretical foundation by **describing FIDO2** in detail. We also review existing projects related to our goal in [Chapter 3](#).

Then, in [Chapter 4](#), we proceed to **the actual implementation**, which represents the main contribution of this project.

Finally, in [Chapter 5](#), we summarize **the achieved results**.

1.2 Project Goal

The goal of this project is to **create a new open-source implementation of an external (roaming) FIDO2 hardware-based authenticator** from scratch that would be *well-documented, thoroughly tested, and production-ready*.

By focusing on the quality and the documentation, this implementation could help others understand the FIDO2 standards by providing a detailed yet accessible insight into the inner workings of these protocols (which the existing implementations lack). In general, this project has the potential to contribute to the popularization of FIDO2 technology.

The result of this project will be an open-source implementation of a **FIDO2 USB hardware external authenticator**. The working of the implementation will be demonstrated on a suitable hardware platform (such as STM32F4). However, the selection of the most appropriate hardware platform and the related code optimizations, which would allow us to use hardware-assisted cryptography, prevent private key extraction, and mitigate some side-channel attacks, are beyond the current project's scope and are left for future work.



Figure 1.1 [Security Key NFC by Yubico](#), one of the most typical FIDO2 USB security keys (or more precisely, cross-platform roaming authenticators).

Chapter 2

FIDO2

In this chapter, we provide a more detailed description of **FIDO2**.

FIDO2 is a set of related specifications, jointly developed by the *FIDO Alliance* and the *W3C (World Wide Web Consortium)*, that together enable **easy, secure, and phishing-resistant authentication** for online services (primarily on the web, but they can be used in native applications as well).

The specifications are:

- **Web Authentication (WebAuthn) API** by the *World Wide Web Consortium (W3C)*
 - This is the core specification that **defines and describes all the key concepts**, some of which we will cover in the following sections.
 - More specifically, it “defines an API enabling the creation and use of strong, attested, scoped, public key-based credentials by web applications, for the purpose of strongly authenticating users” [1].
 - **Level 2** (W3C Recommendation from April 8, 2021) is the latest stable version.
 - **Level 3** is being actively developed and some of its new features (hybrid transport, conditional mediation, hints) are already supported by browsers.
- **Client to Authenticator Protocol (CTAP)** by the *FIDO Alliance*
 - “This specification describes an application layer protocol for **communication between a roaming authenticator and another client/platform**, as well as bindings of this application protocol to different transport protocols” [2].
 - **CTAP 2.1** (Proposed Standard from June 21, 2022) is the latest stable version.
 - **CTAP 2.2** (Review Draft 01) is being actively developed and some of its new features (hybrid transports) are already supported by browsers.
- There is also FIDO U2F (now referred to as CTAP1), which is a predecessor of FIDO2 that can be used only for two-factor authentication (as a second factor). FIDO2 authenticators can be backwards-compatible with CTAP1 (FIDO U2F). However, we will not deal with CTAP1 in our project.

2.1 Functional Description

FIDO2 relies on public key cryptography (also called asymmetric cryptography). [1]

Public key cryptography uses the concept of a key pair. Each key pair consists of a **private key** 🔑 (which must be kept secret) and a corresponding **public key** 🗝️ (which can be openly distributed without compromising security). The public and private keys are mathematically related. They are generated with cryptographic algorithms based on mathematical problems termed *one-way functions*. [15] It is not possible to calculate the private key from the public key. One of the most common public key cryptosystems is RSA that relies on the difficulty of factoring the product of two large prime numbers. [16]

The core idea is that FIDO2/WebAuthn allows servers (i.e., websites, referred to as Relying Parties) to register and authenticate users using public key cryptography.

Instead of a password, a private-public keypair (known as a **credential**) is created for a website during **registration**. While the **private key** 🔑 is stored securely on the user's device (inside the **authenticator**) and it never leaves it, the **public key** 🗝️ and an authenticator-generated credential ID are sent to the server where they are stored.

Then, during **authentication**, the server then uses that **public key** to verify the user's identity by verifying the user's possession of the private key. More specifically, the server sends the credential ID and a random data called a challenge to the user. User's authenticator looks up the **private key** by the credential ID and it signs the challenge data (encrypts their hash, signature = encrypted hash of the data) using the **private key**. It passes this signature back to the server. The server uses the **public key** to verify the signature. It decrypts the hash and compares it with the real hash of the challenge data. If they match, it proves that the user is in possession of the private key and, therefore, can be authenticated.

The **diagram** below illustrates the simplified authentication flow described above:

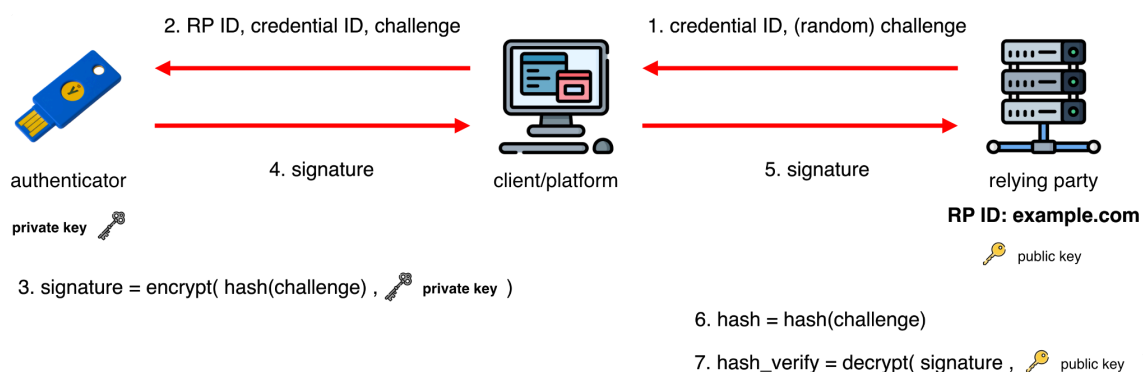


Figure 2.1. The simplified authentication flow. To further clarify, the credentials belong to the user and are managed by an **authenticator**, with which the **Relying Party** interacts through the **client** platform.

2.2 Key Benefits

Based on the description in the [previous section](#), we can clearly see some of the security and privacy benefits of FIDO2 authentication.

- **Security**
 - A new unique public-private key pair (credential) is created for every relying party (and every authenticator the user uses).
 - All credentials are scoped to the relying party (website).
 - The private key never leaves the authenticator.
 - The RP's server does not store any secrets.
 - The possession of the private key is proved by signing a random challenge generated by the RP.
 - This security model **eliminates all risks of phishing, all forms of password theft** (there is **no shared secret to steal**) and **replay attacks**.
- **Privacy**
 - The whole FIDO2 is designed with privacy in mind. All credentials are unique and they are scoped to the relying party (website). They cannot be used to track users in any way.
 - Plus, biometric data, when used for user verification, never leaves the user's device (authenticator).
- **User Experience**
 - Users interact with the authenticator by (typically) simply touching it to provide consent with registration and authentication.
 - Credentials can be further protected and their use might require local user verification, which, based on the authenticator capabilities, might utilize PIN or biometric methods.
 - Users can use cross-platform *roaming authenticators* such as FIDO2 security keys (which communicate via USB, NFC, Bluetooth) or they might rely on their-device built-in *platform authenticators* (such as Apple ID with Face ID or Touch ID, Windows Hello, and Google Password Manager on Android) that are part of the [client device](#).

2.3 Terminology

In the previous sections, we briefly introduced the key parties involved in FIDO2/WebAuthn – *the relying party (RP), the client, and the authenticator*. In this section, we dive into the specifics and provide more detailed information about them and their behavior. We also define additional related terms to aid in our explanations.

2.3.1 Relying party (RP)

“An entity whose application utilizes the WebAuthn API to *register* and *authenticate* users, and which stores the public key.” [1] Typically, such an application has both client-side code to invoke the WebAuthn API on the client (typically in the browser) and server-side code to validate responses and store details about registered credentials (their IDs and public keys) in some sort of database.

2.3.2 Client

An entity (typically a web browser or a similar application) that acts as an intermediary between the *relying party* and the *authenticator*. [1]

2.3.3 Client Device

“The hardware device on which the *client* runs, for example a smartphone, a laptop computer or a desktop computer, and the operating system running on that hardware.” [1]

2.3.4 Client Platform

“A *client device* and a *client* together make up a *client platform*. A single hardware device may be part of multiple distinct client platforms at different times by running different operating systems and/or clients.” [1]

2.3.5 Authenticator

“A cryptographic entity, existing in hardware or software, that can register a user with a given Relying Party” (i.e., generate and store a public-private key pair with appropriate metadata) and “later assert possession of the private key during authentication”. [1] The private keys never leave the authenticator.

The authenticators that are part of the *client device* are referred to as *platform authenticators* (such as Apple ID with Face ID or Touch ID, Windows Hello, and Google Password Manager on Android), while those that are reachable via cross-platform transport protocols (USB, NFC, Bluetooth) are referred to as *roaming authenticators* (such as **FIDO2 USB security keys**).

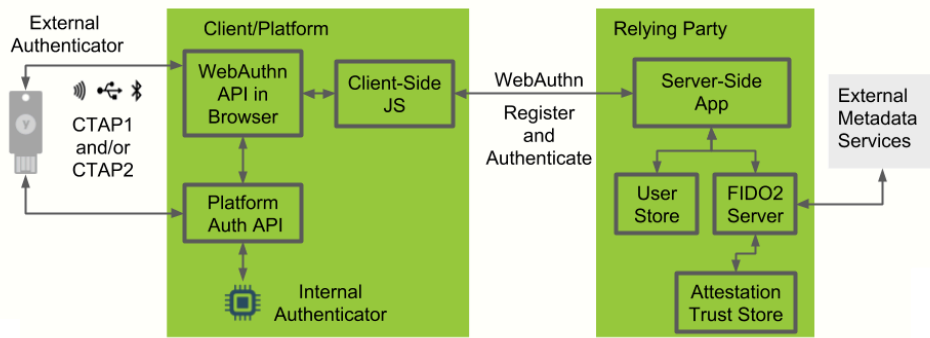


Figure 2.2. The relationship among the key parties in FIDO2. [17] Note that the details of the communication between the client-side part and the server-side part of the application are beyond the scope of FIDO2/WebAuthn, which only mandates that HTTPS or equivalent transport security is used. [1]

2.3.6 Credential Storage Modality

Authenticators (such as FIDO2 hardware security keys) might have a limited storage capacity. Until now, we have assumed that **the private keys** must be stored in the authenticator. In fact, the WebAuthn allows **two different storage strategies**:

1. The private key is stored **in persistent storage** embedded in the authenticator. This strategy is necessary for *client-discoverable/resident credentials* that can be used as a first factor.
2. The private key is **stored within the credential ID**. Specifically, the private key is encrypted (**wrapped**) such that only the authenticator can decrypt (i.e., unwrap) it. The resulting ciphertext is **the credential ID**.

Typically, the authenticator uses symmetric AES encryption, where the symmetric encrypt/decrypt key (sometimes called master key) is securely stored in the authenticator and never leaves it. Therefore, this strategy, if correctly implemented, is as secure as the first one.

Note that this strategy is possible because **the RP passes the credential ID** to the authenticator **during authentication**. Therefore the authenticator does not need to store any information about such a credential in its persistent storage. This way it can support virtually an unlimited number of credentials. However, this strategy **cannot** be used for client-discoverable credentials, where the RP does **not** pass the credential ID to the authenticator and instead the authenticator must be able to list all existing credentials for that RP.

Note that the authenticators **might support both strategies** and use different storage strategies for different credentials (typically only use the persistent storage when it is strictly necessary, i.e., for *client-discoverable* credentials).

For more information, see 6.2.2. [Credential Storage Modality](#) in [1].

Also note that there is a term “passkey”, which is the end-user-centric term for a FIDO2 **client-discoverable** credential.

2.4 Ceremonies

In this section, we build on the information provided in the [Functional Description](#) section, and we cover the **registration** and **authentication** flows in more detail. These flows are referred to as “ceremonies” by the WebAuthn specification because they extend the concept of a computer *communication protocol* with human-computer interactions.

In the following descriptions, we assume a **standard web application** that runs client-side code to invoke the WebAuthn JavaScript APIs in the browser and to communicate with its server, which validates responses and stores/retrieves details about registered credentials (their IDs and public keys). In this case, the browser represents the WebAuthn **client**. Note that WebAuthn can also be used in native applications where platform-specific vendor-provided APIs are used in place of the WebAuthn JavaScript APIs. On Android, these are [FIDO2 API for Android](#) and/or [Credential Manager API](#). iOS provides similar [APIs](#).

2.4.1 Registration

During registration, a new credential is created on an authenticator and registered with a Relying Party server. Registration must happen before a user can use their authenticator to authenticate.

The following [diagram](#) from depicts the **registration** flow:

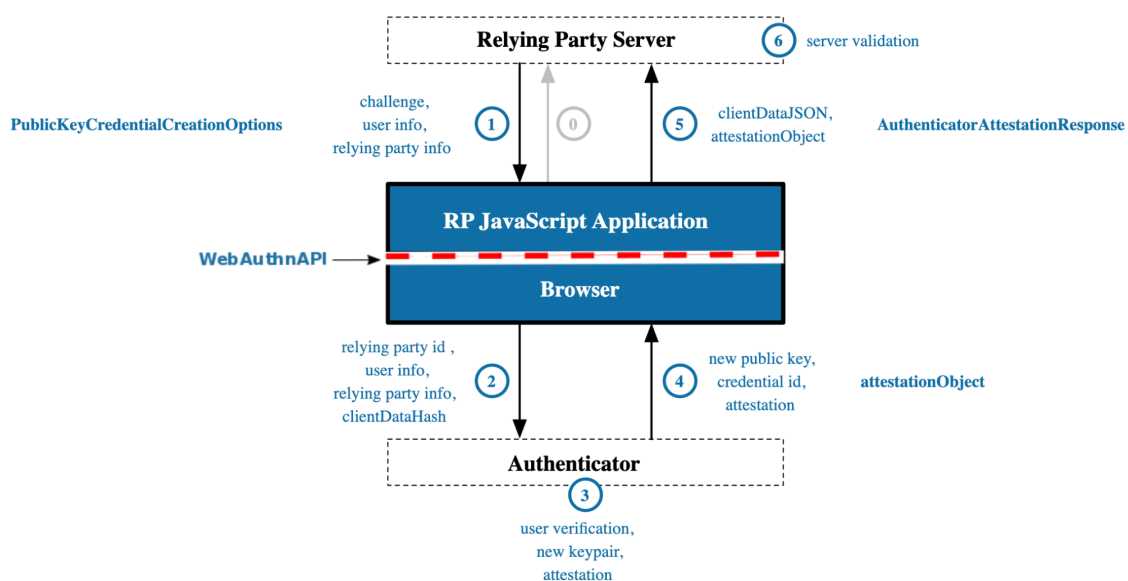


Figure 2.3. The WebAuthn **registration** flow. [1]

The user visits a website, for example, **example.com**, which serves up a script. At this point, the user may already be logged in using a legacy username and password, or additional authenticator, or other means acceptable to the Relying Party. Or the user may be in the process of creating a new account.

0. The user initiates the registration process by interacting with the application in the browser. The application sends a request to the server to start the registration.
1. The server responds with `PublicKeyCredentialCreationOptions` that contains information about the user (its opaque user handle), preferred authenticators (roaming vs platform) and algorithms. It also includes a random data called **challenge**. This is to prevent replay attacks.
2. The client-side code in the browser invokes the WebAuthn API method **`navigator.credentials.create()`** and passes the options from the server. The browser **appends the RP ID** (the website origin, i.e., `example.com`). This is necessary for the **correct scoping** of the credentials to the RP. Also, the browser appends `clientDataHash`. Then, the browser locates a suitable authenticator, establishes a connection to it, and sends the `authenticatorMakeCredential` request. The communication between the browser (the client) and the authenticator uses CTAP2 in case of *roaming authenticators* (e.g., a FIDO USB security key) or some platform-specific communication channel in case of *platform authenticators* (e.g. Windows Hello).
3. The authenticator asks the user for some sort of authorization gesture to provide consent. It may involve user verification (PIN entry or biometric check) or it may only be a simple test of user presence. If the user authorizes the registration, the authenticator **generates a new credential** (a public-private key pair and a credential ID). The private key is securely stored in the authenticator and never leaves it.
4. The authenticator takes **the generated public key** and **the credential ID**, appends the `clientDataHash` and information about itself and signs all this data with either its attestation private key or with the credential private key (so called self-attestation).
5. The client-side code in the browser obtains the response from the authenticator (as the result of the `navigator.credentials.create()` call) and sends it to the server.
6. The server verifies the data. Specifically, it checks the attestation signature. This way, the server ensures that all the credential creation options are respected and that the response is indeed related to the initial request challenge. If everything matches, **the server saves the public key and the credential ID to the database** and associates them with the user.

2.4.2 Authentication

This is the ceremony when a user with an already registered credential visits a website and wants to authenticate using the credential. During authentication, the server then uses that **public key** to verify the user's identity by verifying the user's possession of the private key.

The following **diagram** from depicts the **authentication** flow:

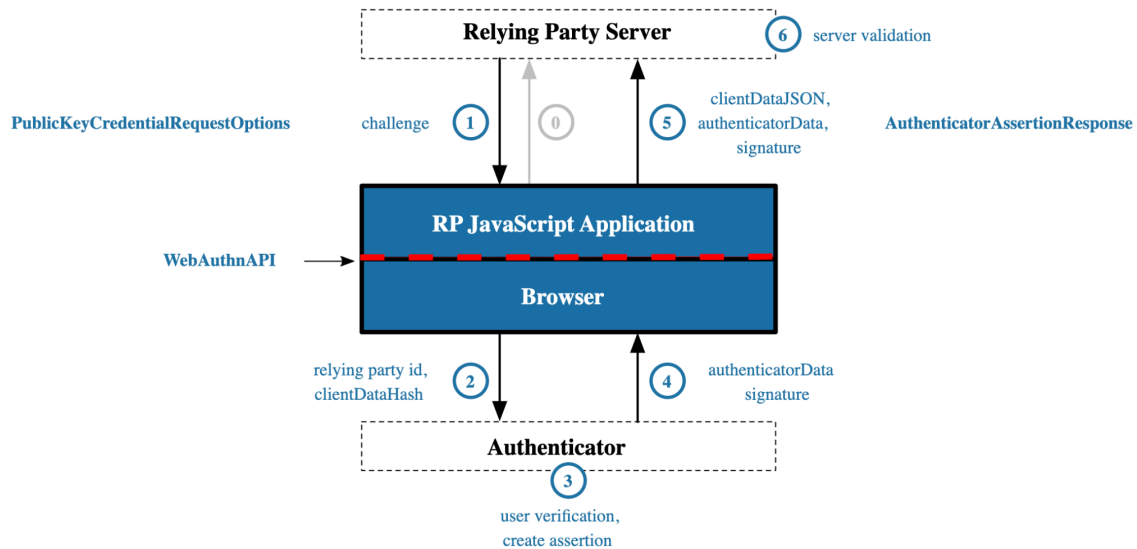


Figure 2.4. The WebAuthn **authentication** flow. [1]

Since we have already described the authentication in sufficient detail in the **Functional Description** section, we omit the detailed description here and ask the reader to refer to the **corresponding description there**.

2.5 Additional Resources

In addition to the specifications [1] and [2], there are a lot of online accessible resources about FIDO2, WebAuthn, and passkeys available. Here are some that we found useful or high-quality:

- **WebAuthn Guide** (webauthn.guide) by Duo
 - a very nice single-page explanation of WebAuthn and related-concepts
- **WebAuthn.wtf** by FusionAuth
 - a very nice overview and multi-page summary of all main WebAuthn concepts
- **WebAuthn Introduction** on Yubico Developers
 - The Yubico (popular hardware security keys vendor) Developers website has multiple resources on [WebAuthn](#) and [passkeys](#).
- The thesis [4] includes a description of FIDO2 (in English) in Chapter 1.2.
- The thesis [3] includes a description of FIDO2 (in Czech) in Chapter 4.

2.6 Security Key Products

There are many hardware security keys from different manufacturers available on the market. Usually, they support communication over USB (either USB-A or USB-C) and NFC.

Below, you can see the two FIDO2 security keys I own and use. They can communicate over USB and NFC (useful for Android and iOS). The **GoTrust Idem Key** boasts **FIDO2 Security Level 2 certification**, which enables it to be used for services that explicitly require it (such as MojeID for the EU/Czech eGovernment services access with the “high” level assurance).



Security Key NFC by Yubico

FIDO2 Security Level 1



GoTrust Idem Key

FIDO2 Security Level 2

FIPS 140-2 Level 3

Chapter 3

Existing Work

Before we started working on our own implementation, we did an extensive search to see what open-source implementations of FIDO2 authenticators were available. The most relevant projects are [SoloKeys](#) and [OpenSK](#).

3.1 Open-Source Software and Hardware

SoloKeys

- “the first open source FIDO2 security key”
- **v1** launched on Kickstarter in [2018](#) and **v2** in [2021](#)
- As of today (June 2, 2024), it seems that **the project development has stopped**, see this [discussion](#). Kickstarter backers of v2 received their keys in 2023.
- **v1**
 - <https://github.com/solokeys/solo1>
 - **software written in C for STM32L432**
- **v2**
 - <https://github.com/solokeys/solo2>
 - **software written in Rust for NXP LPC55S69**

OpenSK by Google

- An open-source implementation for security keys written in **Rust** that supports both **FIDO2** and **FIDO U2F**.
- Actively developed by **Google**.
- Written in Rust, uses [Tock OS](#), the main development target platform is **Nordic nRF52840**.

Nitrokey

- Open Source IT-Security Hardware from a German company
- Nitrokey FIDO2
 - <https://github.com/Nitrokey/nitrokey-fido2-firmware>
 - a fork of **solo1**
- Nitrokey 3
 - <https://github.com/Nitrokey/nitrokey-3-firmware>
 - developed in collaboration with SoloKeys, similar to Solo 2

3.2 Other Relevant Projects

2 bachelor theses at **CTU in Prague** (both from CTU FIT):

- [3] [Matěj Borský \(May 12, 2022\). FIDO2 Authentication Simulator.](#)
 - a simulator of a USB FIDO2 authenticator
 - Linux only, uses [UHID](#) for emulating the authenticator HID device
 - source code **not available** online
- [4] [Martin Kolárik \(June 4, 2020\). FIDO2 KeePass Plugin.](#)
 - **not much relevant to our project**; however, there is a nice overview of FIDO2 in Chapter 1.2
 - deals with the design and development of a plugin for KeePass open-source password manager
 - the plugin should enable the use of a FIDO2 authenticator instead of (in addition to) a master password for unlocking the password database

2 open-silicon designs:

OpenTitan

- OpenTitan: Open source silicon root of trust (RoT) (ASIC / FPGA)
- [OpenTitan Big Number Accelerator \(OTBN\) Technical Specification](#)
- [Ibex RISC-V Core Wrapper](#)
- Based on [this GitHub thread](#) (last activity about a year ago), it seems that someone is working toward getting OpenSK running on the OpenTitan FPGA platform.

TROPIC01 (TRuly OPen IC)

- partially open secure element designed by the Czech company Tropical Square, which is fabless ASIC design spinoff of Satoshi Labs
- the TROPIC01 is designed to be used in Satoshi Labs's crypto HW wallets [Trezor](#)
 - Some of the Trezor HW wallets can be used as a FIDO2 authenticator. See [this about FIDO2](#), [this about U2F](#), and [this comment](#).
- uses <https://github.com/lowRISC/ibex> (small open-source 32-bit RISC-V core, also used in the [OpenTitan](#) project)
- part of the chip responsible for elliptic cryptography developed by a student at CTU FIT – this part is now called SPECT (Secure Processor of Elliptic Curves for Tropic)
 - see this talk [Informatické večery: Kryptografický koprocesor nejen pro bitcoinovou hardwarovou peněženku – FIT ČVUT](#)
 - bachelor thesis (the talk above is more up-to-date): [Víceúčelová hardwarová platforma pro kryptografii nad eliptickými křivkami](#)
 - paper: [Versatile Hardware Framework for Elliptic Curve Cryptography | IEEE Conference Publication](#) (also freely available on CTU DSpace [here](#))

Chapter 4

Implementation

This chapter describes the main parts of our FIDO2 authenticator implementation.

Main points:

- Written in **C**.
- Runs on the **STM3240G-EVAL** board with the **STM32F407IGH6** MCU.
- Uses STM32CubeF4 (its HAL, LL, and USB Device Library) via STM32CubeMX generator.
- FIDO2 (specifically the CTAP2 layer) implementation in **fido2** dir is based on the **SoloKeys Solo 1** project. We **rewritten / refactored / modified / fixed** some parts of the original implementation, so it could be used with our STM32F4 MCU (which has, for example, a different **flash memory organization**). Furthermore, we completely rewritten the **CTAPHID** layer.
- **The complete source code is available on GitHub here:**
<https://github.com/pokusew/fel-krp-project>

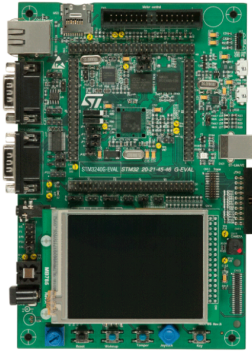


Figure 4.1. The **STM3240G-EVAL** board with the **STM32F407IGH6** MCU.

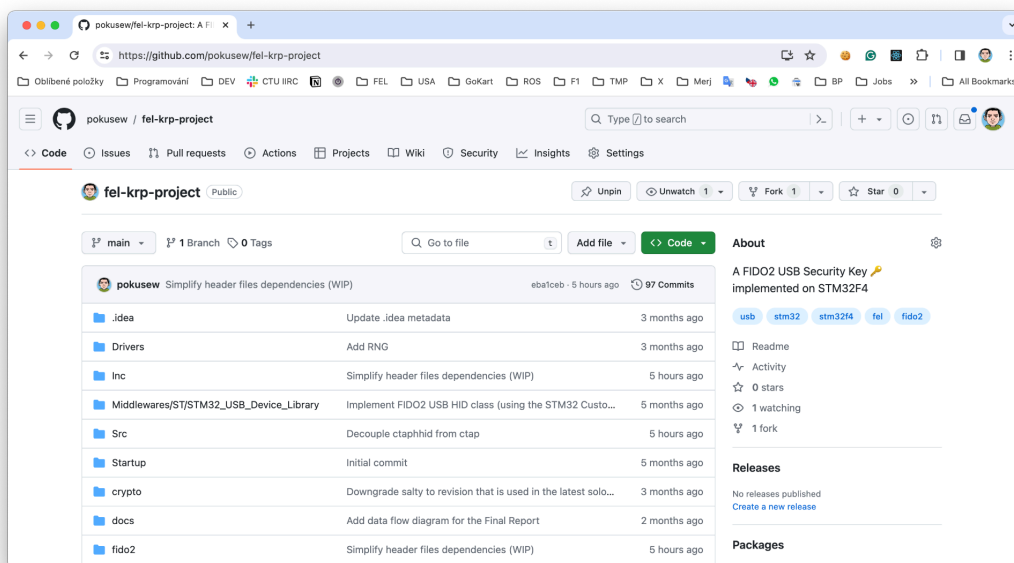


Figure 4.2. The screenshot of the **project repository** homepage on GitHub.

4.1 Architecture

In this section we describe individual layers (protocol stack) of the firmware that together implement the necessary FIDO2 functionality. The [diagram](#) on the right depicts these layers.

4.1.1 CTAP2

The core part is the implementation of the **CTAP2** protocol. CTAP2 protocol is a high-level transaction-oriented protocol. A CTAP2 **transaction** consists of a *request*, followed by a *response message*. CTAP2 transactions are always initiated by the client. Messages are [encoded](#) using the concise binary encoding CBOR.

There are two essential transactions – [authenticatorMakeCredential](#), which is implemented in [ctap_make_credential\(\)](#), and [authenticatorGetAssertion](#), which is implemented in [ctap_get_assertion\(\)](#). They facilitate the [register](#) and [authenticate](#) flows as described in the FIDO2 chapter.

4.1.2 CTAPHID

Mapping of messages to the underlying USB transport is facilitated by the **CTAPHID** layer. Request and response **messages** are divided into individual fragments, known as [packets](#). Packets are the smallest form of protocol data units, which in the case of CTAPHID are mapped into USB **HID reports**. CTAPHID also implements logical channel [multiplexing](#) so that multiple **clients** (browsers, apps, OS) on the same [client device](#) (device) can communicate with the authenticator concurrently.

Our implementation of CTAPHID is described in the [Data Flow](#) section.

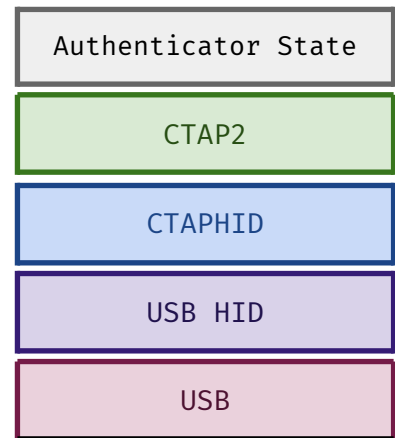


Figure 4.3. Protocol Stack.

4.1.3 USB

The **USB HID** and **USB** layers are implemented using the STM32Cube USB device library and its *Custom HID* class. It uses the USB_OTG_FS (**USB 2.0 Full Speed**) MCU peripheral.

There are **two endpoints (IN, OUT) with interrupt transfer** (64-byte packet max, poll every 5 millisecond). The **descriptors** (device, config, interface, endpoints, **HID report**) are set up according to the **CTAPHID specification (11.2.8. HID device implementation)**.

The **USB HID report descriptor** plays a key role in the automatic *device discovery*. The CTAPHID protocol is designed with the objective of driver-less installation on all major host platforms. Browsers and other clients use the standard USB HID driver included within the OS. The **HID Usage Page** field within the authenticator's HID report descriptor is set to the value 0xF1D0, which is registered to the FIDO Alliance (see **HID Usage Tables 1.5**).

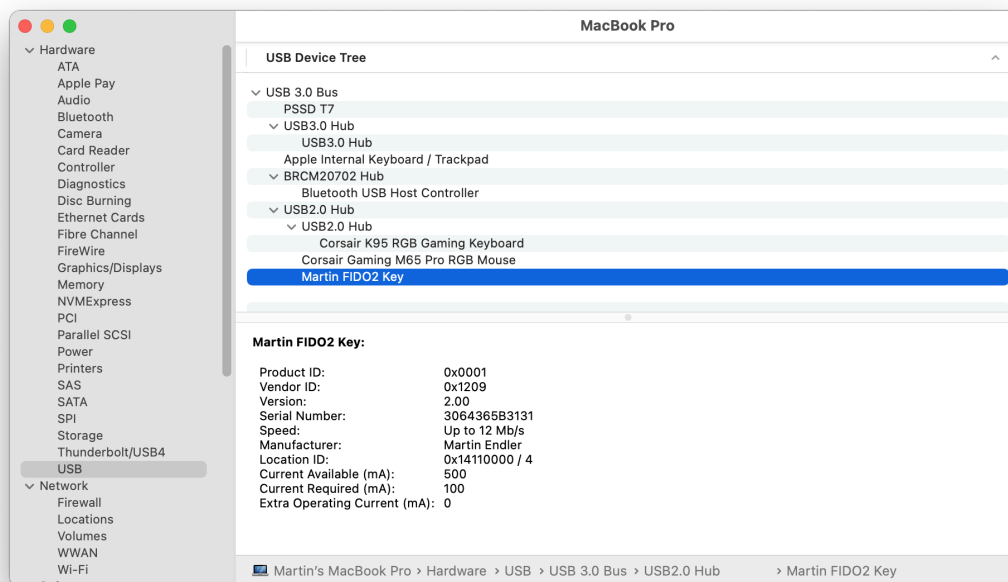


Figure 4.4. The following screenshot shows our authenticator in the USB Device Tree Explorer in System Information on macOS 11.

The chosen Vendor ID (VID) and Product ID (PID) come from the [pid.codes](#) project.

[pid.codes](#) is a registry of USB PID codes for open source hardware projects. They **assign** PIDs on any VID they own to any open-source hardware project needing one.

For the initial version of this project, we used their testing VID/PID **0x1209/0x0001**. In the future, when we have a more-production-ready authenticator, we might eventually **ask for** our own PID.

4.1.4 Authenticator State

The top-most block in the [diagram](#) represents the authenticator and its internal state.

4.1.4.1 Persistent Storage

The most complex part is the implementation of **state persistence**. There are multiple pieces of data that need to be persisted. Namely, it is:

- the master key (used for storing [non-discoverable credentials](#), see more info in [Credential Storage Modality](#)),
- the PIN (or the information that it is not has not been set yet) for user verification (UV) and invalid PIN attempts counter (note that the PIN is not stored in plaintext, instead it is stored as a salted hash),
- the global [signature counter](#),
- and client-side discoverable credentials (formerly called resident credentials, the abbreviation rk, which is used in the code, stands for **resident key**).

Note: This document uses the following notation: 1 KB (kilobyte) = 1024 B (bytes).

As a persistent storage, we used **a part of the 1024 KB of Flash memory** that is built into the STM32F407IGH6 MCU. By default, it is used for storing the program code (firmware). Because our firmware takes only about the first 104540 B (~ 102 KB), i.e., around 10% of the total capacity, we can use the rest for our application data.

However, flash memory comes with **an important limitation** – one can only program (write) flash bits from 1 to 0. In order to write 1 in place of the already written 0 bits, one must **erase the whole memory sector** (the size of which can be in the range of KBs). This is generic to flash memory technology and is not specific to the STM32.

In case of STM32F407IGH6, its flash memory organization is as follows (also documented in [Inc/flash.h](#)):

A main memory block divided into 12 sectors
(with **different** sizes, numbered 0–11):

- 4 sectors of 16 KB: sectors 0–3
- 1 sector of 64 KB: sector 4
- 7 sectors of 128 KB: sectors 5–11

Our program occupies the sectors 0–4 (102 KB fits into the first 128 KB). So, we are only left with the big 128KB sectors for the application data. **Their erasure takes a significant amount of time** (seconds). Also, the total possible number of erasures is limited (**flash memory wear**). Thus, it is not possible to naively rewrite the whole sector when only a few bytes need to be changed (for example, when the global signature counter needs to be incremented).

There are special file systems designed to allow efficient use of such memories. However, for our use case, **we implemented a simpler, custom solution, adapted from the Solo 1 project** (where they faced the same challenge, but their MCU's sectors were only 1 KB big which still allowed for relatively fast erasures).

The implemented algorithms are conceptually types of **simple wear-leveling algorithms**. With our solution, the need for erasures is greatly reduced. Most of the time, all the changes in the authenticator state can be done without any erasure at all (so it takes no extra time).

Here are the relevant parts of the code that implement the persistence:

- flash memory utils in [Inc/flash.h](#) and [Src/flash.c](#)
- [Inc/memory_layout.h](#) that defines the used sector numbers
- [ensure_flash_initialized\(\)](#) in [Src/app.c](#) for initialization of the memory layout upon the very first startup and after memory reset (using the [app_delete_data](#) debug command)
- persistence of the [AuthenticatorState](#) struct (contains the master key + PIN data) with the **simple wear-leveling algorithm** to allow fast changes:
 - [authenticator_read_state\(\)](#)
 - [authenticator_write_state\(\)](#)
- persistence of the global [signature counter](#)
 - [ctap_atomic_count\(\)](#) in [Src/device.c](#)
 - Note: A special effort has been made to make the counter *atomic*. In case the flash write is interrupted (e.g., the authenticator is disconnected from the USB and the power is interrupted), the counter should keep its previous value. This is critical for the correct FIDO2 implementation because the counter must never be decremented.
- persistence of the client-discoverable credentials (resident credentials, resident keys)
 - This is the only part where the wear-leveling algorithm is not implemented yet.
 - Fortunately, it has only a negligible impact, since the [ctap_overwrite_rk](#) is only called once when a new client-discoverable credential is being created. This is in contrast with [ctap_atomic_count\(\)](#) and [authenticator_read_state\(\)](#) that can be invoked even multiple times during a single transaction.
 - [ctap_overwrite_rk\(\)](#)
 - [ctap_reset_rk\(\)](#)
 - [ctap_rk_size\(\)](#)
 - [ctap_store_rk\(\)](#)
 - [ctap_delete_rk\(\)](#)
 - [ctap_load_rk\(\)](#)

■ 4.1.4.2 True Random Number Generator

We used the RNG peripheral to generate true random numbers for cryptographic purposes.

The RNG peripheral available in the STM32F407IGH6 MCU is a random number generator, based on a continuous analog noise, that provides a random 32-bit value when read.

The RNG passed the FIPS PUB 140-2 (2001 October 10) tests with a success ratio of 99%.

The relevant parts of the code:

- `rng_get_bytes()` in `Src/rng.c`

4.2 Data Flow

When a new USB HID report packet (in the code we also call it HID msg) arrives from the host to the authenticator, the function `CUSTOM_HID_OutEvent_FS` is invoked (within the interrupt context). It takes the HIT report (64 bytes) and puts it in the `hidmsg_buffer` circular buffer (FIFO) that has capacity of $100 * 64$ bytes.

The `main app loop` regularly checks the buffer and if there is an unprocessed message (report), it passes it to the CTAPHID layer's `ctaphid_handle_packet` function. The CTAPHID layer reconstructs the CTAP2 messages from the individual packets. When a CTAP2 **request** message is ready to be processed it is passed to the CTAP2 layer via the `ctap_handler`. After the CTAP2 layer processes the request, the response message is sent (it is split into individual USB HID packets (reports) which are sent one by one by `usbhid_send`).

The incoming data path (from the host to the authenticator) is visualized in the following diagram:

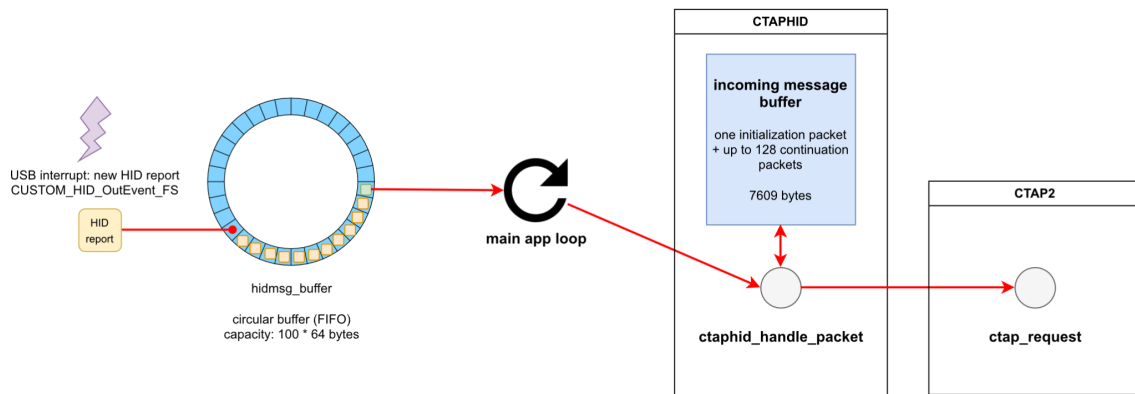
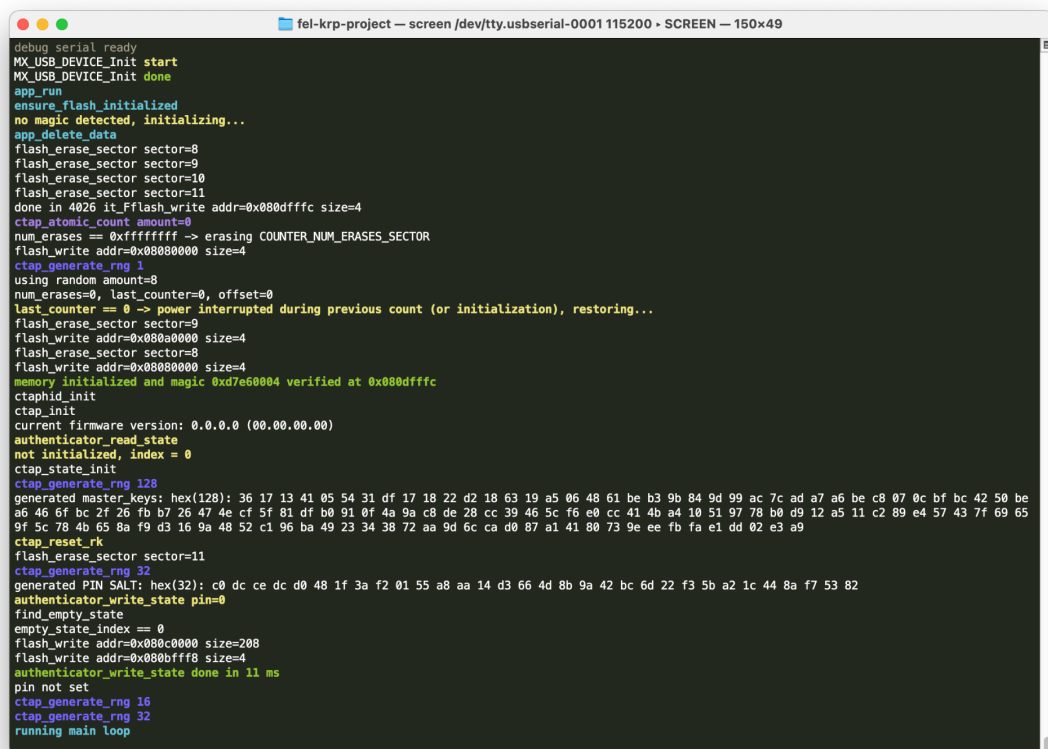


Figure 4.5. The incoming data path (from the host to the authenticator).

Note that when checking the `hidmsg_buffer` (or taking data from it), the interrupts (or at least the USB interrupt) **must be disabled** to prevent race conditions (because new data could be written at any time by the `CUSTOM_HID_OutEvent_FS` that runs in the interrupt context).

4.3 Debugging Interface via UART

In order to allow easy debugging (and development of the USB communication stack), we implemented a simple **bidirectional** debugging interface using UART (USART3 peripheral). We wrote a simple `write syscall` implementation so that the `printf` statements (wrapped in custom `debug/info/error_log` macros) could work as expected. The handling of the incoming debug commands is done in the app main loop [here](#).

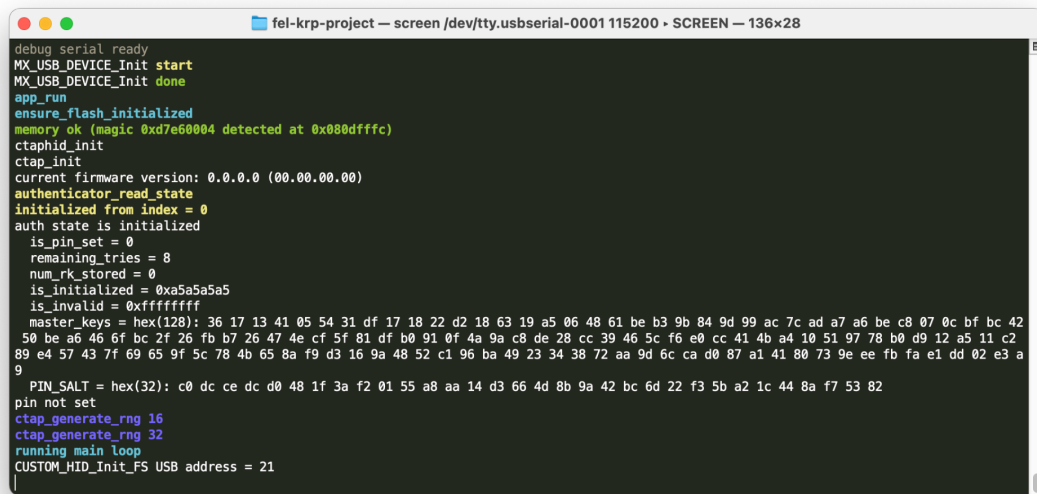


```
fel-krp-project — screen /dev/tty.usbserial-0001 115200 • SCREEN — 150x49
debug serial ready
MX_USB_DEVICE_Init start
MX_USB_DEVICE_Init done
app_run
ensure_flash_initialized
no magic detected, initializing...
app_delete_data
flash_erase_sector sector=8
flash_erase_sector sector=9
flash_erase_sector sector=10
flash_erase_sector sector=11
done in 4026 it fflash_write addr=0x080dffc size=4
ctap_atomic_count amount=0
num_erases == 0xffffffff -> erasing COUNTER_NUM_ERASES_SECTOR
flash_write addr=0x08000000 size=4
ctap_generate_rng 1
using random amount=8
num_erases=0, last_counter=0, offset=0
last_counter == 0 -> power interrupted during previous count (or initialization), restoring...
flash_erase_sector sector=9
flash_write addr=0x08000000 size=4
flash_erase_sector sector=8
flash_write addr=0x08000000 size=4
memory initialized and magic 0xd7e60004 verified at 0x080dffc
ctaphid_init
ctap_init
current firmware version: 0.0.0.0 (00.00.00.00)
authenticator_read_state
not initialized, index = 0
ctap_state_init
ctap_generate_rng 128
generated master_keys: hex(128): 36 17 13 41 05 54 31 df 17 18 22 d2 18 63 19 a5 06 48 61 be b3 9b 84 9d 99 ac 7c ad a7 a6 be c8 07 0c bf bc 42 50 be
a6 46 6f bc 2f 26 7b b7 26 47 4e cf 5f 81 df b0 91 0f 4a 9a c8 de 28 cc 39 46 5c f6 e0 cc 41 4b a4 10 51 97 78 b0 d9 12 a5 11 c2 89 e4 57 43 7f 69 65
9f 5c 78 4b 65 8a f9 d3 16 9a 48 52 c1 96 ba 49 23 34 38 72 aa 9d 6c ca d0 87 a1 41 80 73 9e ee fb fa e1 dd 02 e3 a9
ctap_reset_rk
flash_erase_sector sector=11
ctap_generate_rng 32
generated PIN SALT: hex(32): c0 dc ce dc d0 48 1f 3a f2 01 55 a8 aa 14 d3 66 4d 8b 9a 42 bc 6d 22 f3 5b a2 1c 44 8a f7 53 82
authenticator_write_state pin=0
find_empty_state
empty_state_index == 0
flash_write addr=0x080c0000 size=208
flash_write addr=0x080bfff8 size=4
authenticator_write_state done in 11 ms
pin not set
ctap_generate_rng 16
ctap_generate_rng 32
running main loop
```

Figure 4.6. A screenshot of the initial debug logs upon authenticator's first startup when the memory has not been initialized yet.

You can notice that the authenticator generates a new master key upon its first startup. This master key is used for [non-discoverable credentials](#).

On subsequent startups, the initialization is skipped when initialized memory is detected.



```
debug serial ready
MX_USB_DEVICE_Init start
MX_USB_DEVICE_Init done
app_run
ensure_flash_initialized
memory ok (magic 0xd7e60004 detected at 0x080dffc)
ctaphid_init
ctap_init
current firmware version: 0.0.0.0 (00.00.00.00)
authenticator_read_state
initialized from index = 0
auth state is initialized
is_pin_set = 0
remaining_tries = 8
num_rk_stored = 0
is_initialized = 0xa5a5a5a5
is_invalid = 0xffffffff
master_keys = hex(128): 36 17 13 41 05 54 31 df 17 18 22 d2 18 63 19 a5 06 48 61 be b3 9b 84 9d 99 ac 7c ad a7 a6 be c8 07 0c bf bc 42
50 be a6 46 6f bc 2f 26 fb b7 26 47 4e cf 5f 81 df b0 91 0f 4a 9a c8 de 28 cc 39 46 5c f6 e0 cc 41 4b a4 10 51 97 78 b0 d9 12 a5 11 c2
89 e4 57 43 7f 69 65 9f 5c 78 4b 65 8a f9 d3 16 9a 48 52 c1 96 ba 49 23 34 38 72 aa 9d 6c ca d0 87 a1 41 80 73 9e ee fb fa e1 dd 02 e3 a
9
PIN_SALT = hex(32): c0 dc ce dc d0 48 1f 3a f2 01 55 a8 aa 14 d3 66 4d 8b 9a 42 bc 6d 22 f3 5b a2 1c 44 8a f7 53 82
pin not set
ctap_generate_rng 16
ctap_generate_rng 32
running main loop
CUSTOM_HID_Init_FS USB address = 21
```

Figure 4.7. A screenshot of the startup debug logs when the memory is already initialized.

There are **several debug commands**, each one of them is assigned to one **lowercase** letter:

- **l** (lowercase L) – toggle the Blue LED
- **t** – **app_test_time**
- **g** – **app_test_rng**
- **f** – **app_flash_info**
- **c** – **app_test_ctap_atomic_count**
- **s** – **app_test_state_persistence**
- **d** – **app_delete_data**

This command deletes all data (client-discoverable credentials, global counter) that are stored in the persistent flash memory. After that, Reset is needed upon which the memory will be reinitialized.

The four on-board LEDs are used as status indicators:

- **Green** – main loop running
- **Orange** – turned on USB device has been initialized and assigned an address
- **Blue** – toggle using UART debug char l (lowercase L)
- **Red** – ErrorHandler
 - This indicates that some unexpected fatal error occurred. Press the reset button. The device will not work until reset.

Chapter 5

Conclusion

In this project, **we successfully implemented** a working FIDO2 USB hardware external authenticator (also called FIDO2 USB security key).

First, we outlined the goal of the project and its motivation. Then, we established the necessary theoretical foundation by describing the Web Authentication and CTAP2 standards, which are together known as FIDO2. We also reviewed existing projects related to our goal.

Then, we proceeded to the actual implementation, which represents the main contribution of this project. We documented our decisions and the most important parts of the implementation. All code is versioned using Git and is publicly **available online on GitHub**. The repository includes build and run instructions. There is also a **CI/CD pipeline** to ensure software quality. Our authenticator supports both client-discoverable credentials and an unlimited number of non-discoverable credentials. Thanks to the full support of client-discoverable credentials and user verification (UV) using PIN, it can be used as a first factor (passkey).

Finally, **we tested** our authenticator **with real websites** with different configurations (1st factor vs 2nd factor). In all cases, **it worked great**, including with Google Account and GitHub.

However, the authenticator is not production-ready. Even though we were able to make it work (with the help of the Solo 1 codebase), there are a lot of details and edge cases that are not correctly handled. The whole FIDO2/WebAuthn specification is very complex and a proper implementation would take much more time.

5.1 Future Work

There are a few more steps needed to achieve the ultimate goal of creating a new FIDO2-compliant implementation from scratch that would be thoroughly tested and production-ready.

Currently, our implementation uses a part of the Solo 1 project (the CTAP2 layer) that we need to replace with our own implementation.

Furthermore, we would like to use test-driven development and ensure there is sufficient test coverage. As of now, we have set up a **CI/CD pipeline** (so at least we can test that project builds correctly), but there are no unit/integration/e2e tests yet.

Ultimately, we would like to attempt to pass the FIDO **Authenticator Level 1 (L1) certification**.

References

[1] W3C (April 8, 2021). Web Authentication: An API for accessing Public Key Credentials – Level 2.

W3C Recommendation.

<https://www.w3.org/TR/webauthn-2/>

Note: There are newer working versions of this standard, see the links in the document.

[2] FIDO Alliance (June 21, 2022). Client to Authenticator Protocol (CTAP).

CTAP 2.1 Proposed Standard.

<https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-errata-20220621.html>

Note: See <https://fidoalliance.org/specifications/download/> for the list of the current published standards. Namely, there is already a newer version available (CTAP 2.2 Review Draft 01).

[3] Matěj Borský (May 12, 2022). FIDO2 Authentication Simulator.

<https://dspace.cvut.cz/handle/10467/101916>

<https://dspace.cvut.cz/bitstream/handle/10467/101916/F8-BP-2022-Borsky-Matej-theis.pdf>

[4] Martin Kolárik (June 4, 2020). FIDO2 KeePass Plugin.

<https://dspace.cvut.cz/handle/10467/88264>

<https://dspace.cvut.cz/bitstream/handle/10467/88264/F8-BP-2020-Kolarik-Martin-theis.pdf>

[5] Data Breach Investigations Report 2020

<https://www.verizon.com/business/en-gb/resources/reports/2020-data-breach-investigations-report.pdf>

Additional info: <https://www.verizon.com/business/resources/reports/dbir/>

[6] Bitwarden World Password Day Survey (2024)

<https://bitwarden.com/resources/world-password-day/#overview>

[7] FIDO Alliance – Overview

<https://fidoalliance.org/overview/>

[8] FIDO Alliance – FAQ

<https://fidoalliance.org/faqs/>

[9] FIDO Alliance – User Authentication Specifications Overview

<https://fidoalliance.org/specifications/>

[10] FIDO Alliance – Member Companies & Organizations

<https://fidoalliance.org/members/>

[11] FIDO Alliance May 5, 2022). Apple, Google and Microsoft Commit to Expanded Support for FIDO Standard to Accelerate Availability of Passwordless Sign-Ins

<https://fidoalliance.org/apple-google-and-microsoft-commit-to-expanded-support-for-fido-standard-to-accelerate-availability-of-passwordless-sign-ins/>

[12] passkeys.dev – Device Support

<https://passkeys.dev/device-support/>

[13] MDN Web Docs – Web Authentication API – Browser compatibility

https://developer.mozilla.org/en-US/docs/Web/API/Web_Authentication_API#browser_compatibility

[14] Can I use... – Web Authentication API

<https://caniuse.com/webauthn>

[15] Public-key cryptography – Wikipedia

https://en.wikipedia.org/wiki/Public-key_cryptography

[16] RSA (cryptosystem) – Wikipedia

[https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem))

[17] developers.yubico.com – WebAuthn Introduction

<https://developers.yubico.com/WebAuthn/>