FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	FIDO2 KeePass Plugin
Student:	Martin Kolárik
Supervisor:	Ing. Jiří Dostál, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2020/21

Instructions

The goal of this thesis is to implement a plugin for the official KeePass version for Windows that allows unlocking a KeePass database using a FIDO2 device instead of the regular master password/key file method. If possible, the plugin should preserve database compatibility with other clients, i.e., it should be possible to unlock the database using the regular master password/key file method if the user configured it instead of the FIDO2 device.

1. Analyze the possibilities provided by the KeePass plugin system and review the implementations of existing key provider plugins.
2. Design a suitable approach to implementing a key provider plugin utilizing a FIDO2 device, focusing on security, ease of use and portability.
3. Implement, test and document the plugin.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D. Head of Department	doc. RNDr. Ing. Marcel Jiřina, Ph.D. Dean

Prague October 31, 2019

**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

Bachelor's thesis

FIDO2 KeePass Plugin

Martin Kolárik

Department of Software Engineering
Supervisor: Ing. Jiří Dostál, Ph.D.

June 4, 2020

Acknowledgments

I would like to thank my supervisor Ing. Jiří Dostál, Ph.D. for his interest in this topic and the provided support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on June 4, 2020 ....

Czech Technical University in Prague
Faculty of Information Technology
© 2020 Martin Kolárik. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Kolárik, Martin. *FIDO2 KeePass Plugin*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020. Also available from: ⟨https://github.com/MartinKolarik/KeePassFIDO2⟩.

Abstrakt

Táto práca skúma možnosti využitia FIDO2 zariadení ako náhradu pre hlavné heslo v správcoch hesiel. Popisuje možnosti a obmedzenia FIDO2 a diskutuje možnosti implementácie: ukladanie šifrovacích kľúčov na FIDO2 autentifikátore a použitie autentifikátora na zašifrovanie externe uložených kľúčov. Časť o implementácii popisuje zvolený postup ukladania šifrovacích kľúčov na autentifikátore, problémy súvisiace s obmedzením prístupu k FIDO2 zariadeniam v aktuálnych verziách Windows a architektonické rozhodnutia na prekonanie týchto problémov. Ako celok je táto práca užitočným zdrojom informácií pre kohokoľvek pokúšajúceho sa využiť FIDO2 v správcovi hesiel alebo podobnom prostredí.

Kľúčové slová FIDO2, WebAuthn, KeePass, správca hesiel, bezpečnostný kľúč, prihlásenie bez hesla

Abstract

This thesis explores the possibilities of using FIDO2 devices as a replacement for master passwords in password managers. It describes the capabilities and current limitations of FIDO2, and discusses implementation approaches: storing encryption keys on FIDO2 authenticators, and using the authenticators to encrypt externally stored keys. The implementation chapter describes the chosen approach of storing encryption key on the authenticator, the encountered challenges with restrictions on access to FIDO2 devices in recent versions of Windows, and the architectural decisions made to overcome those challenges. As a whole, the thesis is a useful reference for anyone attempting to utilize FIDO2 in password managers or similar environments.

Keywords FIDO2, WebAuthn, KeePass, password manager, security key, passwordless login

Contents

Introduction **1**

1 State-of-the-art **5**
1.1 KeePass password manager 5
1.1.1 KeePass plugin system 6
1.1.2 KeePass key providers 8
1.2 FIDO2 10
1.2.1 Web Authentication (WebAuthn) 12
1.2.2 Client-to-Authenticator Protocol (CTAP) 14
1.2.3 Universal 2nd Factor (U2F) 15
1.2.4 A note on terminology 15

2 Analysis and design **17**
2.1 Design goals 17
2.2 Implementation options 20
2.2.1 Key stored as an authenticator resident key 20
2.2.2 Key encrypted by the authenticator 20
2.2.3 Encryption key derived from a signature 21
2.2.4 Key stored in resident key metadata 25

3 Implementation **27**
3.1 Communicating with the authenticator 28

3.2 Plugin architecture 29

4 Testing **33**

4.1 Positive tests 34

4.1.1 Add a key for a new database 34

4.1.2 Unlock a database using the authenticator 34

4.1.3 Unlock a database using the original method 35

4.2 Negative tests 35

4.2.1 No database open 35

4.2.2 Error while adding a new key 36

4.2.3 Error while unlocking the database 36

Conclusion **37**

Bibliography **39**

A Acronyms **43**

B Glossary **45**

C Contents of the enclosed SD card **49**

List of Figures

1.1 A minimal KeePass plugin 7
1.2 A minimal KeePass key provider 9
1.3 FIDO2 architecture 11

2.1 OpenPGP message encryption 21
2.2 A signature scheme with appendix 22

3.1 The plugin options window 30
3.2 The database unlock dialogue 31
3.3 Sequence diagram of a get() operation 31

List of Tables

1.1 Comparison of DLL and PLGX plugin format 7

Introduction

Passwords have been in use as a primary method of authentication in computer systems for decades. They are a well-understood concept by both the engineers implementing those systems and by the end-users.

Still, they come with several problems. With the computational power of today's hardware, they need to be long and complex to be secure, which makes them hard to remember and annoying to type. The problem with remembering becomes significantly worse upon the realization that an average person may use tens of different services, each of which should be protected by a different password. Of course, it is almost impossible to remember that many passwords, which is why most people tend to use just one, or a few passwords everywhere. This practice presents a significant security risk, because compromising just one system may lead to compromising all other systems where the same password is used by the given user and could be seen as one of the main arguments for moving to a different form of authentication.

There have been many attempts to find a suitable replacement over the years and predictions of passwords going away. For example, Bill Gates made such a prediction already in 2004, and in 2011, IBM predicted that

"you will never need a password again" within five years. Yet, it is 2020, and passwords are still the primary method of authentication in most systems. None of the possible replacements, such as digital certificates, one-time login links, biometrics, or single sign-on systems, managed to combine the relative ease-of-use, security, cost-effectiveness, and ease of implementation and deployment.

Since passwords have been in wide use for a long time, solutions that aim not to replace them, but rather remove some of their problems, were developed. Password managers—applications designed to securely store all user's passwords in an encrypted form—solve the issue with passwords reuse by removing the need to remember them, and may even provide better user experience by automatically filling in the correct credentials in most environments. They are usually protected by another password themselves—a master password—which is the only password the user needs to remember.

The idea of replacing passwords altogether has not been lost, though, and one of the most recent news in this area is FIDO2. A project by the FIDO Alliance—an open industry association with members including many large technology companies, such as Amazon, Apple, Google, or Microsoft—and World Wide Web Consortium—an organization responsible for creating standards for the World Wide Web.

FIDO2 uses authenticators—small devices similar to flash keys—which are able to securely generate, store, and later find and use the correct credentials for each service, while completely hiding the technical aspects from the end-users.

This approach provides several advantages over passwords, including complete elimination of phishing attacks, while keeping a great level of usability—from the users' perspective, using the device involves only connecting it to the computer and unlocking it with a short PIN when prompted.

The current issue with FIDO2 is that being a very new technology—designated as an official web standard in March 2019—even if it becomes successful, it will take years before the majority of online services supports it and before the whole ecosystem around it develops.

Nevertheless, having a large list of big organizations behind it, and being focused on all the aforementioned problems of alternatives (ease-of-use, security, cost-effectiveness, and ease deployment), it has great potential.

The goal of this thesis is, therefore, exploring the possibilities of combining the FIDO2 technology with passwords managers to provide a solution that works today, and makes it easy to gradually move beyond passwords, as the support for FIDO2 increases.

Specifically, the thesis analyzes the capabilities of FIDO2 and KeePass password manager (similar concepts should apply to other passwords managers as well), discusses the options of replacing master passwords with FIDO2 devices, and provides a proof-of-concept implementation.

Chapter 1

State-of-the-art

As a first step, we are going to take a look at how KeePass and FIDO2 work individually. After that, we can examine the options of combining them.

1.1 KeePass password manager

KeePass is a free open source password manager licensed under GNU General Public License (GPL). It stores passwords, and all associated data, such as usernames, URLs, notes, etc. in a single *database* file, which is encrypted using a *master key*. The supported encryption algorithms include Advanced Encryption Standard (AES) and ChaCha20 [1].

The master key is based on one or more *key sources*:

- a master password,
- a key file, which can be any file, located anywhere on the system or external storage,
- a windows user account, in which case Windows Data Protection API (DPAPI) is used.

Any of these sources may be used on its own or be combined with others [2].

KeePass is written primarily for the Windows operating system and can be run on other systems via Mono or Wine [3]. There are many unofficial KeePass ports to other platforms, including Android, iOS, Linux, and OS X. These ports usually aim to implement the same database format, i.e., they might be able to use a database created by the official version, but the overall set of features and implementation details may differ considerably [4]. This work will focus only on the official version but will try to preserve database compatibility with other versions, if possible.

1.1.1 KeePass plugin system

KeePass offers an extensive plugin system, and a list of known plugins is maintained at its official website [5]. Looking at the categories of available plugins provides a good overview of what the plugin system allows:

- I/O & Synchronization – loading or synchronizing a database from a remote storage provider,
- Integration & Transfer – improving integration with other applications for easier transfer of credentials,
- Cryptography & Key Providers – adding new encryption methods or key sources,
- Import & Export – interoperability with other systems.

This is, of course, not an extensive list of the plugins system's capabilities.

The plugins are authored in C#, using the .NET Framework, and need to derive from a base KeePass plugin class, which defines hooks for KeePass lifecycle events as seen in Figure 1.1 [6].

```
using KeePass.Plugins;

namespace SimplePlugin
{
    public sealed class SimplePluginExt : Plugin
    {
        private IPluginHost m_host = null;

        public override bool Initialize(IPluginHost host)
        {
            if (host == null) return false;
            m_host = host;
            return true;
        }
    }
}
```

Figure 1.1: A minimal KeePass Plugin [6]

The `IPluginHost` interface provides access to most of KeePass's internals. Several sample plugins are available, demonstrating the most common use-cases, such as:

- creating a custom password generator algorithm,
- creating a custom encryption algorithm,
- customizing the UI elements [5].

The plugins can be distributed as a DLL file, a PLGX[1] file, or both [6]. The advantages and disadvantages of these options are summarized in Table 1.1.

Table 1.1: Comparison of DLL and PLGX plugin format [6]

	DLL	PLGX
Compatibility check	No - weak only	Yes - strong
Compatibility with custom builds (Linux)	Partial	Yes
Authenticode signing support	Yes	No
No compilation on the user's system	Yes	No
No plugin cache	Yes	No

[1]An optional plugin file format for KeePass $\geq$ 2.09. Instead of compiling the plugin to a DLL file, the plugin source code files are packed into a PLGX file and KeePass compiles them itself when the plugin is first loaded [6].

1.1.2 KeePass key providers

In the previous section, we found there is a category of plugins called key providers, which allow implementing alternative ways of unlocking the database. Key providers derive from the `KeePassLib.Keys.KeyProvider` class and register themselves in the key provider pool via the `Add` method of the `KeyProviderPool` class provided by `IPluginHost` interface [7], as seen in Figure 1.2. This design follows a strategy pattern[2] that allows users to select which of the registered providers will be used at database creation time.

In this section, we will briefly examine some of the existing implementations. This is not meant to be an extensive list of key providers nor a detailed examination of their implementation details, but rather a small selection of well-documented, actively developed, and used[3] open source implementations that we can use as a reference later when discussing our own implementation challenges.

KeePassQuickUnlock is a plugin that allows reopening the database without a full master password. It only works if the database is closed and later reopened, but KeePass stayed running—in that case, the full master key is kept in memory, and only a few characters of the master password are required to confirm the unlock [12].

Similarly to KeePassQuickUnlock, KeePassWinHello is intended for quickly unlocking the database—after its first regular unlock—using Windows Hello[4] technology. By default, the plugin holds an encrypted master key in memory and removes it when KeePass is closed. In order to be able to unlock

[2] A behavioral software design pattern that allows us to define a family of algorithms, make them interchangeable, and select a specific one at runtime [8].

[3] The KeePass website lists several plugins implementing support for RSA certificates, for example, which might be conceptually similar to our plugin, but after a short inspection, we found that they are either no longer available [9, 10], or not actively developed and used [11] (judging by a lack of author and user activity, and little or no documentation).

[4] A technology that adds alternative way to authenticate into Windows and applications using a fingerprint, iris scan, facial recognition, a short PIN, or other method.

the database via Windows Hello in between KeePass launches, it may be configured to store the key in the Windows Credential Manager[5] [13].

```
namespace KeyProviderTest
{
    public sealed class KeyProviderTestExt : Plugin
    {
        private IPluginHost m_host = null;
        private SampleKeyProvider m_prov = new SampleKeyProvider();

        public override bool Initialize(IPluginHost host)
        {
            m_host = host;
            m_host.KeyProviderPool.Add(m_prov);
            return true;
        }

        public override void Terminate()
        {
            m_host.KeyProviderPool.Remove(m_prov);
        }
    }

    public sealed class SampleKeyProvider : KeyProvider
    {
        public override string Name
        {
            get { return "Sample Key Provider"; }
        }

        public override byte[] GetKey(KeyProviderQueryContext ctx)
        {
            // Return a sample key. In a real key provider plugin, the key
            // would be retrieved from smart card, USB device, ...
            return new byte[]{ 2, 3, 5, 7, 11, 13 };
        }
    }
}
```

Figure 1.2: A minimal KeePass key provider [7]

[5] A place where Windows and other apps using its API store credentials scoped to a specific Windows account.

1.2 FIDO2

FIDO2 is a term used to refer to several related specifications, which together, "enable users to leverage common devices to easily authenticate to online services in both mobile and desktop environments" [14]. The specifications are the Web Authentication (WebAuthn) specification by World Wide Web Consortium (W3C) and the Client-to-Authenticator Protocol (CTAP) specification by FIDO Alliance[6].

The security model of FIDO2 is based on public-key cryptography, and three main entities:

- a *relying party (RP)* – an entity whose application utilizes the WebAuthn API, and which stores the public key,
- an *authenticator* – a cryptographic entity that handles generating and storing keys, and performing cryptographic operations,
- a *client* – an entity that acts as an intermediary between the relying party and the authenticator (typically a web browser or a similar application) [15].

For each relying party, a separate public and private key pair is used, instead of a regular password.

The authenticator may be either a separate hardware device (a *roaming authenticator*), or a platform implementation (a *platform authenticator*). The main difference is that while a roaming authenticator is transferable between *client devices*[7], a platform authenticator is bound to a specific client device. A platform authenticator may still use specialized hardware, e.g., a Trusted Platform Module (TPM)[8] to provide sufficient security [15].

[6] An open industry association focused on authentication standards that aim to reduce the use of passwords. Members include many large technology companies, such as Amazon, Apple, Google, or Microsoft.

[7] A hardware device on which the client runs, e.g., a smartphone or a laptop.

[8] A secure crypto-processor that is designed to carry out cryptographic operations and store cryptographic keys.

Note that authenticators are often also referred to as “security keys”. We will restrain from using this term in this work to avoid confusion with frequently mentioned encryption keys.

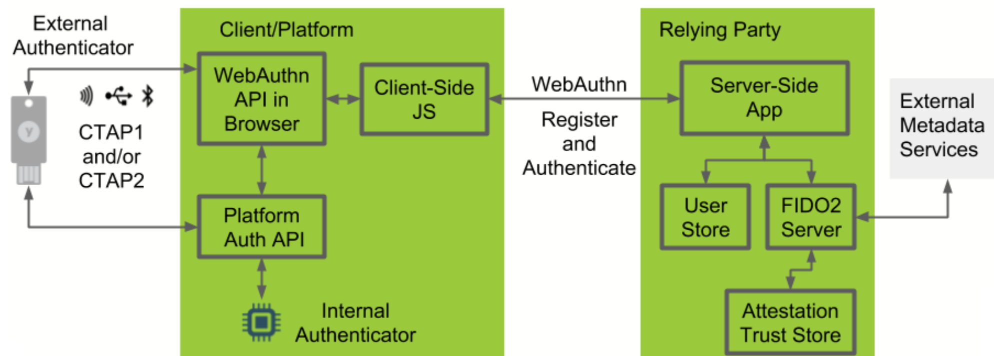

Figure 1.3: FIDO2 architecture [16]

The FIDO Alliance summarizes the key aspects of FIDO2 as [14]:

1. **Security** – FIDO2 cryptographic login credentials[9] are unique across every website, never leave the user’s device, and are never stored on a server. This eliminates the risks of phishing, all forms of password theft, and replay attacks.

2. **Privacy** – Because FIDO cryptographic keys are unique for each internet site, they cannot be used to track users across sites.

3. **Convenience** – Users unlock cryptographic login credentials with simple built-in methods such as fingerprint readers or cameras on their devices, or by leveraging easy-to-use FIDO security keys.

4. **Scalability** – Websites can enable FIDO2 through a simple JavaScript API call that is supported across leading browsers and platforms.

[9]The term “credential” is not well-defined by the specification, but generally means the private key and its associated metadata stored by the authenticator, such as information about the user account or relying party.

1.2.1 Web Authentication (WebAuthn)

WebAuthn is the specification that covers interactions between clients and relying parties (typically, but not necessarily, web browsers and web applications). The main interactions are `Create` and `Get`, which serve to create new credentials and to authenticate using existing ones.

`Create` can be loosely described as [15]:

1. The relying party (optionally) collects user data, sets credential creation options, and initiates the create process with the client.

2. The client forwards the request to the authenticator, along with information about the relying party.

3. The authenticator generates a new credential and returns the public key along with additional metadata to the client.

4. The client forwards the key and metadata to the relying party.

There are two creation options that are especially important in this process, `userVerification` and `requireResidentKey`, which specify whether user verification is required, preferred, or discouraged, and whether a resident credential is requested.

User verification is a process by which the authenticator locally authorizes the invocation of its operations. User verification may be instigated through, for example, a touch plus pin code, password entry, or biometric recognition [15]. In other words, this is the element that prevents unauthorized use of the authenticator by people other than its owner who have physical access to it. This protection is optional and can be enforced by the relying party. When enabled, it takes place before step three.

Resident credential is a credential whose private key is stored in the authenticator, client, or client device. This definition immediately raises a question: where else could the private key be stored? The answer is that

this is not specified—the authenticator is responsible for protecting the key, but not for its physical storage. This allows implementing schemes that offload the storage to an external system so that the number of stored keys is not limited by the authenticator's internal memory. For example, Yubico[10] uses the following process for handling non-resident keys [17]:

> During credential registration, a new key pair is randomly generated by the YubiKey, unique to the new credential. The private key, along with some metadata about the credential, is encrypted using authenticated encryption with a master key. This master key is unique per YubiKey, generated by the device itself upon first startup, and never leaves the YubiKey in any form [...]
>
> The encrypted (and authenticated) data then forms the 64-byte key handle, which is sent to the server as part of the registration flow, to be stored by the RP for later [...]
>
> For authentication, the RP returns the key handle to the YubiKey. Here it is decrypted to re-form the private key which is needed to sign the challenge to complete the authentication.

The resident credential does not necessarily provide better security, but it comes with one significant advantage that allows simplifying the authentication process. First, let us examine how the `Get` operation works with non-resident credentials [15]:

1. The relying party asks the user for a username or another identifier of the account.
2. Based on the username, it finds all public keys and their metadata that it previously stored.

[10] A member of the FIDO Alliance and a vendor of YubiKey authenticators.

3. The relying party generates a *challenge*, a randomly generated piece of data that the authenticator is expected to sign.
4. The relying party initiates the get process with the client and sends it the information about public keys and the challenge.
5. The client forwards this request to the authenticator.
6. The authenticator performs user verification if requested.
7. The authenticator looks up the private keys corresponding to the public keys provided by the relying party, signs the challenge using each of these keys, and returns the result to the client.
8. The client forwards the signatures and metadata to the relying party.

The resident credentials allow skipping the first two steps, i.e., the user does not need to provide any account information. The authenticator looks up all credentials associated with the relying party. If more than one exists, either the authenticator or the client may present a list of all found credentials and let the user choose which one should be used. If there is only one matching credential, no user input is needed.

1.2.2 Client-to-Authenticator Protocol (CTAP)

This specification describes the communication between clients and authenticators, and exact steps how authenticators handle the individual commands. The commands are closely tied to those already described in WebAuthn so it is enough to say that the `Create` operation from WebAuthn corresponds with the `authenticatorMakeCredential` command, and the `Get` operation is based on `authenticatorGetAssertion`. We may reference this specification in later sections for specific details if they turn out to be important for this work.

1.2.3 Universal 2nd Factor (U2F)

U2F is an open standard for two-factor authentication, which can be seen as a predecessor to FIDO2. Unlike FIDO2, it is already supported by many[11] services, and all FIDO2 devices are backward-compatible with existing U2F implementations [19]. We will not directly utilize the U2F protocol in this work but consider it important to note that since FIDO2 authenticators support U2F as well, it can be used as an addition to passwords for services that implement U2F but not FIDO2.

1.2.4 A note on terminology

Because FIDO2 supports various authentications flows, e.g., it can be used as:

- a password replacement,
- a second factor, in addition to a regular username and password,
- a password and a username replacement, in case of a resident credential,

and because it is also backward-compatible with U2F, it is often not clear which of these flows a specific implementation or service supports. The term FIDO2, when used in this work, always refers to a scenario where the authenticator is used as a primary factor (a replacement for the password). For scenarios where the authenticator is used as an additional factor, we use the term U2F.

[11]The Works with YubiKey Catalog [18] lists tens of large-scale services, such as Dropbox, Gmail, Facebook, or Twitter. It is not a complete list as services do not need to register with Yubico in order to use U2F.

Chapter 2

Analysis and design

In this chapter, we formulate the design goals that our implementation should meet and then discuss four possible implementation approaches.

2.1 Design goals

After analyzing the KeePass plugin system and FIDO2 capabilities, we can now formulate how the combination of KeePass and a FIDO2 authenticator could look in more detail:

1. Websites that support FIDO2 can use this form of authentication exclusively, without passwords.
2. For websites that do not support FIDO2, a regular password stored in KeePass can be used.
   a) For websites that support the older U2F, the FIDO2 device can be used for additional security as a second factor.
   b) The KeePass database can be unlocked by the FIDO2 device instead of a master password.

Our implementation will, therefore, be a KeePass key provider plugin to make the point 2b possible.

An important fact to keep in mind is that KeePass does not support keys being used alternatively, i.e., when a database is configured to be protected by a master password and a custom key provider, they are *both* required to unlock it. It is not possible to configure it, at least by the end-user, in such a way that the unlock methods could be used interchangeably [2].

This is not ideal because introducing a custom key provider means the database cannot be unlocked on other clients unless they also implement the same key provider, and one of our goals was preserving database compatibility with other clients.

There are, however, ways to bypass this restriction. The QuickUnlock plugin introduced earlier does precisely that—first, the database is unlocked using the full master key, and after that, only by a few of its characters.

When we examine its implementation closely, we can find that [12]:

1. When a database is created, the QuickUnlock plugin is *not* registered as a key provider. Instead, the user uses a regular master password or a key file.

2. Once the database is unlocked, the QuickUnlock plugin reads the database encryption key, which is available via the provided plugin interface, and stores this key in an encrypted form.

3. The next time user is unlocking the database, they select the QuickUnlock provider. The QuickUnlock provider decrypts the previously stored master key using the short password entered by the user and unlocks the database with this master key.

This approach means that while from the user's point of view, the database is being unlocked by the QuickUnlock plugin, the used key is identical with the one KeyPass would derive from the master password, which means the QuickUnlock and master password unlock method may be used interchangeably.

We can use a similar approach:

1. When creating a database, the user chooses the primary key—a master password, a key file, or even another method implemented by another plugin.

2. After the database is created, the user will have an option to associate a FIDO2 authenticator with it. In this step, our plugin will retrieve the encryption master key and store it in a secure way.

3. During the next unlock, the user may choose our provider, and the master key will be retrieved and used.

This approach means:

- The user can always decide whether they want to use the primary unlock method or our plugin.

- Multiple authenticators may be associated with one database.

- If only one authenticator is associated and it gets lost, the primary method works as a backup.

Someone might object that one of our goals was to remove the usage of a master password, and this approach still requires it. It is, however, important to note that if the database is only used on systems where our plugin is available, the master password is never needed. In theory, users who do not want to have anything to remember, can create a database with a long and random password (which can be generated using the KeePass generator), associate an authenticator with it right after, and then they do not need to remember the password. The downside is that there is no backup unlock method in case they lose the authenticator, so this would only be advisable if at least two authenticators are associated with the database.

2.2 Implementation options

We have established that we want to take the existing master key, and store it, while "protecting" it by the authenticator. Now is the time to look at the technical possibilities of doing do.

2.2.1 Key stored as an authenticator resident key

The authenticator was designed to securely store private keys, so of course, the first idea might be: can we take an existing key—the one generated by KeePass—and transfer it to the authenticator?

The answer is no, unfortunately, as the authenticator was designed to handle the whole process—including generating the credentials—on its own, and there is no interface that would allow storing externally generated credentials [20].

2.2.2 Key encrypted by the authenticator

Since we need to generate a new key pair on the authenticator, maybe we could use the private key from that pair to encrypt the database master key, and then store the encrypted master key in the unencrypted section of the database header section [21].

This design, where an asymmetric key is only used to encrypt a symmetric key, and the symmetric key is then used to encrypt and decrypt the data, can be found, for example, in OpenPGP[12] as seen in Figure 2.1 [22].

[12] An email encryption standard defined by the OpenPGP Working Group of the Internet Engineering Task Force (IETF) as a Proposed Standard in RFC 4880.

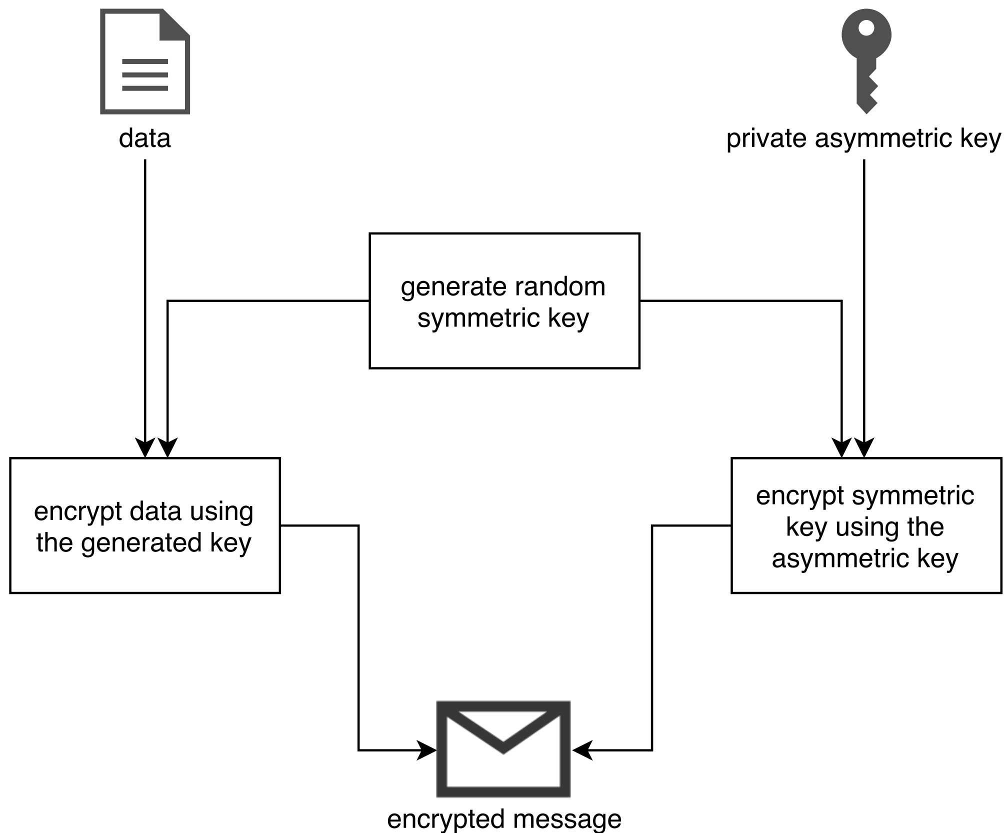

Figure 2.1: OpenPGP message encryption

However, after inspecting the CTAP specification, we find—similarly to the previous case—that while the authenticators are likely technically capable of encrypting data[13], it is not something they were designed to do, so this functionality is not exposed over their API [20].

2.2.3 Encryption key derived from a signature

Following the previous ideas, we know that:

- we cannot store the master key as a credential directly on the authenticator,

[13]Considering that signing data typically involves the same underlying operations as encrypting it, and that authenticators may utilize encryption as a way to implement unlimited storage for non-resident credentials, as described in subsection 1.2.1.

- we cannot use the authenticator directly to encrypt the key and store it elsewhere.

Given these constraints, let us take a close look at the one operation the authenticators support—signing data. The WebAuthn specification does not specify the exact algorithm to be used. The authenticator may implement any number of algorithms from the IANA CBOR Object Signing and Encryption (COSE) Algorithms Registry [23] and pick one based on preferences expressed by the relying party [15]. Generally, though, the commonly used signature schemes:

1. take input data to be signed and a private key,
2. apply some cryptographic operations on these two (the operations depend on the exact algorithm being used),
3. output a "signature" [24].

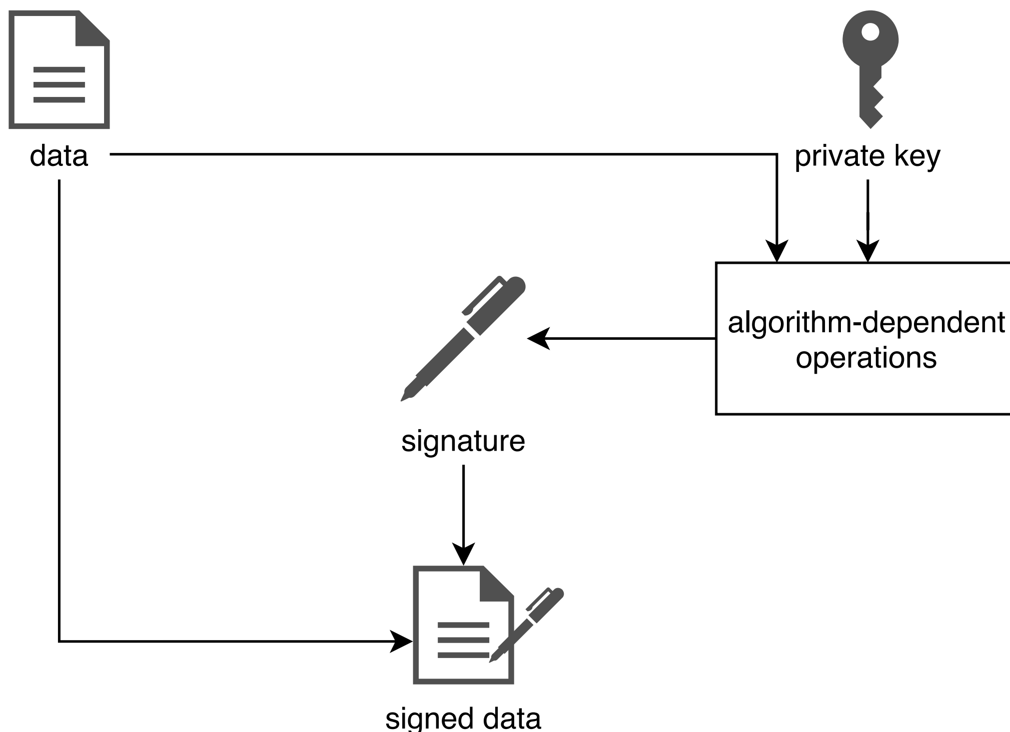

Figure 2.2: A signature scheme with appendix

The signature can be characterized by a few essential properties:

- a user can efficiently produce their own signature on documents of their choice,
- other users can efficiently verify whether a given string is a signature of another (specific) user on a specific document,
- it is infeasible to produce signatures of other users to documents that they did not sign [25].

Usually, the purpose of a signature is to prove the authenticity of the signed data, i.e., the signature is considered "public", but the fact that it is based on the private key, and cannot be created without its knowledge begs the question: could the signature itself act as a key in an encryption scheme?

If yes, we might be able to do the following:

1. Generate a resident credential on the authenticator.
2. Generate random data to be signed (challenge) and store it in the database header section.
3. Ask the authenticator to sign the challenge using the generated private key.
4. Encrypt the database master key using the signature as an encryption key.
5. Store the encrypted database master key in the database header section.

To unlock the database without the master password, we would then:

1. Take the stored challenge from the database header section.
2. Ask the authenticator to sign it.

3. Use the signature to decrypt the encrypted master key from the database header section.

4. Use the database master key to unlock the database.

Of course, this is not a typical usage of cryptographic primitives, so it brings several questions.

First, we need to make sure that repeatedly performing the sign operation using the same challenge and the same credential always generates the same signature, since we use the signature for decryption. If it changed, we would not be able to decrypt the master key. This is something that is not guaranteed, as it depends on the chosen signature algorithm. We can, however, make a list of algorithms that have this property and request that one them is used (and accept that the plugin will work only if the authenticator implements at least one such algorithm).

Note that the WebAuthn specification says that the challenges must be randomly generated by relying parties to prevent replay attacks[14]. This requirement is based on the intended use-case—authentication—but is in direct contradiction with the properties needed for encryption, as explained in the previous paragraph.

The second question is security. We know we will treat the signature itself as something "private", and not store it anywhere. We also know it is created based on a private key stored on the authenticator (which we assume to be secure), using one of standardized signature algorithms (which we also assume to be secure[15]), and that the private key will not be used for any other purpose. For now, let us assume that this scheme is secure, but if this implementation option is eventually chosen, this should be discussed in more detail.

[14] A form of attack in which a valid data transmission is repeated by an adversary who intercepts and re-transmits it.

[15] In a sense that the generated signature has the previously introduced properties.

2.2.4 Key stored in resident key metadata

We have already established that we cannot import existing private keys to the authenticator. However, the credentials are not just private keys—there are several other fields, which may hold additional data, and can be set by the user at the credential creation time. They are defined as [20]:

1. **User handle** – Specified by a relying party, and used to map a specific credential to a specific user account with the relying party. A user handle is an opaque byte sequence with a maximum size of 64 bytes, not meant to be displayed to users, and should not contain personally identifying information.

2. **Display name** – A Human-palatable[16] identifier for a user account, intended only for display, for example, “Alex P. Müller”. Authenticators must accept and store up to at least 64 bytes long values.

3. **Name** – Similar to display name, used to distinguish between user accounts with similar display names. For example, “alex.p.mueller @example.com”. Authenticators must accept and store up to at least 64 bytes long values.

4. **Icon** – A URL that resolves to an image associated with the entity. For example, user’s avatar or a relying party’s logo. Authenticators must accept and store up to at least 128 bytes long values. Note that icon value can employ “data” URLs as defined by RFC2397 [27] so that the icon can be displayed without an internet connection.

The specifications also have important notes about access to these additional fields:

1. “User identifiable information (name, DisplayName, icon) MUST not be returned if user verification is not done by the authenticator” [20].

[16] Intended to be rememberable and reproducible by typical human users, in contrast to identifiers that are, for example, randomly generated sequences of bits [26].

2. “The user handle is not considered personally identifying information [...] It is RECOMMENDED to let the user handle be 64 random bytes” [15].

This is interesting because even though each field has a specified use-case, handling of its content is entirely up to the client and the relying party. Combined with the fact that three of those fields are expected to contain user identifiable information and are protected the same way as private keys, we essentially get a way to store at least 256 bytes of data per credential in a secure way.

This allows us to formulate another design approach. When generating a new resident credential, pass the database master key as an `icon` field of the credential, and keep it stored on the authenticator. To later unlock the database, the contents of the icon field can be retrieved after successful user verification.

Note that we could use the `name` or `display name` field as well. The `icon` was chosen because it provides the most storage space and because it is meant to be displayed as an image. Hence, its raw byte value is least likely to be directly displayed to the user in case the credential is accessed by a client other than our plugin. Of course, a successful user verification would still be required by the authenticator in such a case, so this is not meant to be a significant security mechanism—it could be compared to the common practice of hiding passwords behind asterisks to prevent shoulder surfing.

Also, note that this approach is not ideal. It should be secure, and we include it to document all considered options, but it still requires bending the specification rules and using the `icon` field in a way it was not designed to.

Chapter 3

Implementation

This chapter describes the final part of this work—creating a proof-of-concept implementation of the plugin. Initially, we chose the third implementation option described in subsection 2.2.3. However, we later discovered that our initial analysis missed one crucial detail. The section 6.1.1 of WebAuthn specification introduces an optional *signature counter* [15]:

> Authenticators SHOULD implement a signature counter feature. The signature counter is incremented for each successful authenticatorGetAssertion operation by some positive value, and its value is returned to the WebAuthn Relying Party within the authenticator data. The signature counter's purpose is to aid Relying Parties in detecting cloned authenticators [...]

If an authenticator implements the signature counter, the counter's value is included in metadata that are added to the relying party's challenge before signing it. Supporting this feature is optional, but if an authenticator does support it, there is no way to prevent its use [15]. That means the resulting signature is different every time, which breaks one of the essential requirements of our implementation strategy. After this discovery, we were, therefore, only left with the last option described in subsection 2.2.4.

3.1 Communicating with the authenticator

Rather than communicating with the authenticator directly via CTAP, we initially chose to use libfido2 package by Yubico. It handles communication with a FIDO device over USB and exposes all its features as both a C library and a command-line tool [28].

This choice turned out to be a problem later, as starting with a Windows 10, version 1903, Microsoft has restricted the direct access to FIDO devices to privileged applications. Applications not running with administrator privileges have to use a new native API [28].

Requiring administrator privileges to use our plugin would limit its usability and might raise security concerns, so we attempted to switch to the native API. This has, however, brought several new problems.

First, the only available piece of documentation for this API is a C header file in one of Microsoft's GitHub repositories [29]. Further, the README file in this repository points back to the WebAuthn and CTAP specifications "for more details", but the exposed interfaces from the header file do not directly match those described in the specifications.

Eventually, we have found that the API does not expose the full functionality to applications that use it. For example, when an authenticator returns multiple credentials, Windows uses its own graphical interface to let the user choose which credential they want to use. In this interface, it displays the credential metadata (`name` and `display name`). It does not, however, expose any metadata to the application.

Because our implementation requires access to the `icon` field, it cannot use the native Windows API, and we have to accept it will only work when running under a privileged account.

At this point, it is fair to say that FIDO2—at least in its current state—did not turn out to be a good choice for the purpose of implementing an alternative database unlocking strategy. Out of the four examined implementation approaches, we have found that three will not work at all (for the first two, the reasons are explained directly in the initial analysis, for the third one, at the beginning of this chapter), and the fourth approach—which was also not ideal from the beginning—will require administrator privileges to use our plugin, which might make it impossible to use for some people, create security issues, and overall make the plugin harder to use. In the next section, we describe a way to partially limit the negative aspects of this requirement, but it is not possible to do so entirely.

3.2 Plugin architecture

In this section, we describe the architecture of the plugin, implementing the approach proposed in subsection 2.2.4. The chosen architecture consists of two modules:

- the KeePass plugin itself, written in C# and implementing the required interface,
- a "device communicator" module, in the form of a native executable file written in C++, which performs the operations that require privileged access.

This choice was made for two main reasons. First, because KeePass plugins run in the context of the main KeePass process, performing the privileged operations directly in the plugin would mean the whole KeePass application —and in turn, any other loaded KeePass plugin—would have to be running with administrator privileges, breaking the principle of least privilege[17].

[17]This principle says that every program should operate using the least set of privileges it needs to function, which limits the damage that can result from an accident or programmer error [30].

Second, because libfido2 is an unmanaged DLL written in C, we would need to create a wrapper class acting as an intermediary between the unmanaged DLL, and the managed C# code [31].

By implementing the device communicator as a separate executable, we allow the plugin—and consequently, KeePass—to run under a regular account and greatly reduce the amount of code running in a privileged context.

The plugin module extends the KeePass user interface by adding a new entry "KeePassFIDO2 Options" to the "Tools" menu. After selecting this entry, the user can associate a new authenticator with the currently open database. To do so, they need to click the corresponding button and perform the regular user verification, as requested by the authenticator. The options window is shown in Figure 3.1.

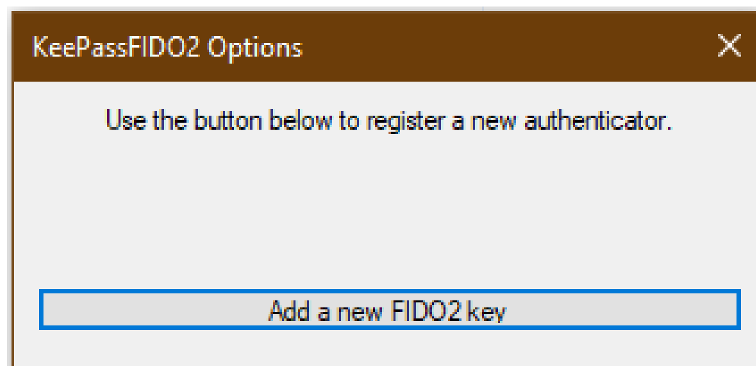

Figure 3.1: The plugin options window

Any number of authenticators can be associated with a single database, but for now, it is not possible to use the same authenticator with multiple databases. This is a limitation that could be removed in future plugin versions by generating a unique database file identifier, storing it in the database header section and on the authenticator, and using it to pair a specific database file with the correct credential.

After adding the key, the database can be unlocked by selecting "FIDO2

Key Provider" in the KeePass unlock dialogue, as shown in Figure 3.2.

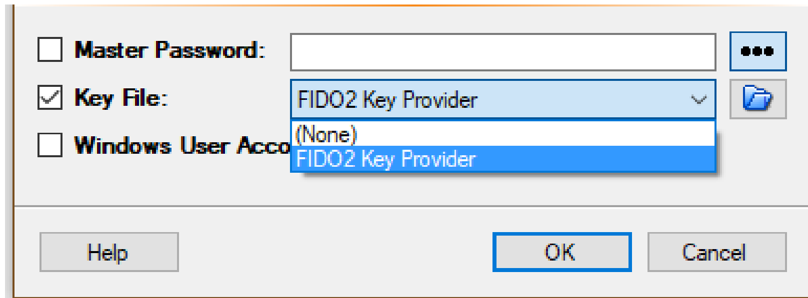

Figure 3.2: The database unlock dialogue

The communicator module implements two operations. The first is `create` to create a new credential, and the second is `get` to retrieve a key stored in an existing credential. These operations are exposed over a minimal interface that allows passing of the necessary data between the two modules. Figure 3.3 shows how retrieving a stored key looks in this architecture.

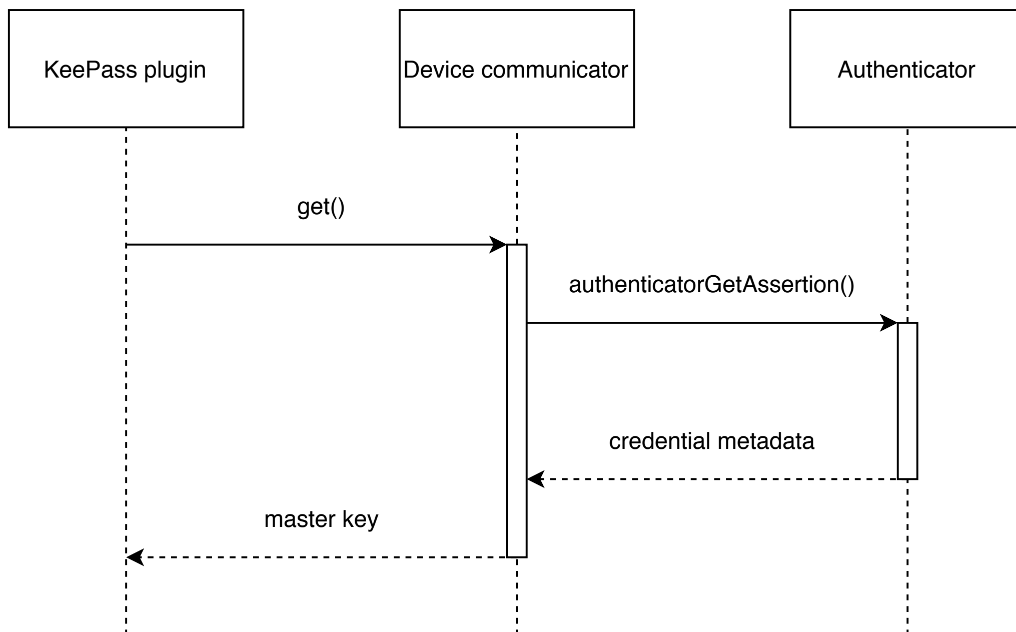

Figure 3.3: Sequence diagram of a get() operation

Because the exchanged data include the authenticator PIN and the database encryption key, the primary criteria for choosing the communication channel were security and ease of implementation.

The plugin allocates a buffer where it prepares the PIN and then passes a pointer to this buffer along with the requested operation name to the communicator via command line arguments. The communicator reads the prepared buffer, verifies it has the required structure, and in case of the `get` operation, later uses it to pass the retrieved key back to the plugin.

Note that the buffer is allocated within the private memory space of the plugin. This approach takes advantage of the fact that the communicator is already running as a privileged process and can access the memory of other processes. Other running unprivileged processes are not able to access this memory.

In the end, this implementation fulfills our initial goal of creating an alternative database unlock strategy using a FIDO2 device, as well as the design goals described in section 2.1. Users can associate any number of authenticators with their KeePass database, and interchangeably use any of them or the original unlock method. Associating an authenticator with a database file does not modify it in any way so that it can still be used with other clients implementing the KDBX file format.

During our work, we used Security Keys by Yubico. The plugin should work with any other USB authenticator implementing the specifications but was tested only with the Yubico keys.

An unstated goal of this work was open-sourcing the final implementation and making it freely available for all KeePass users. Given the previously described issues, however, it should be considered a proof-of-concept only and may not be suitable for widespread use.

Chapter 4

Testing

Testing is an important part of the software development process, even more so in case of software handling sensitive data. The core of our implementation, however, is based on using external hardware devices, which were specifically designed to confirm every operation with the user before performing it. That makes it impossible to write fully automated tests. Writing automated tests only for the parts that do not directly interact with the authenticator, or mocking[18] the functions that perform the communication makes little sense because such tests would not cover the most critical parts of the code.

For these reasons, we include test cases for manual testing, which can be easily performed by any user of the plugin. These test cases are written in such a way that they can also serve as a user manual.

All cases assume a default configuration of Windows 10, version 1903 or higher (mainly, default UAC settings), default KeePass configuration, and a single authenticator connected via USB. All cases start with KeePass running and no database open.

[18] A technique of replacing one part of real implementation with a “fake” version during testing, with the goal of making another part easier to test.

Note that at this time, a single authenticator can only hold a key for one database. Adding a key for a new database (as done in some of these tests) overwrites any other key previously stored on the authenticator.

4.1 Positive tests

This section describes test cases for the intended usage scenarios. Note that these cases are written for a database protected by a master password. To test with a database protected by a key file, the same steps apply, except any instruction to enter the password is changed to choose the key file.

4.1.1 Add a key for a new database

1. Create a new KeePass database.
2. In the “Create Composite Master Key” dialogue, fill in “123” as master password.
3. Go to Tools → Options → KeePassFIDO2 and click “Add a new FIDO key”.
4. Enter your authenticator PIN when prompted (if configured), and click “Submit”.
5. Confirm the UAC prompt for “DeviceCommunicator.exe”.
6. Perform user presence/verification check as requested by the authenticator.

Expected result: A message confirming the key was added appears in the “KeePassFIDO2 Options” window.

4.1.2 Unlock a database using the authenticator

1. Open the database created in subsection 4.1.1.

2. In the “Open Database” dialogue, select “Key File: FIDO2 Key Provider” as unlock method and click “OK”.
3. Enter your authenticator PIN when prompted (if configured), and click “Submit”.
4. Confirm the UAC prompt for “DeviceCommunicator.exe”.
5. Perform user presence/verification check as requested by the authenticator.

Expected result: The database was unlocked.

4.1.3 Unlock a database using the original method

1. Open the database created in subsection 4.1.1.
2. In the “Open Database” dialogue, select “Master Password”, type “123” and click “OK”.

Expected result: The database was unlocked.

4.2 Negative tests

This section describes steps to test that common error situations are correctly handled.

4.2.1 No database open

1. With no database open, go to Tools → Options → KeePassFIDO2 and click “Add a new FIDO key”.

Expected result: An error message is shown, saying that a database needs to be open first.

4.2.2 Error while adding a new key

1. Open the database created in subsection 4.1.1.
2. In the “Open Database” dialogue, select “Master Password”, type “123” and click “OK”.
3. Go to Tools → Options → KeePassFIDO2 and click “Add a new FIDO key”.
4. Disconnect the authenticator from the computer.
5. Submit the “FIDO2 PIN” dialogue.
6. Confirm the UAC prompt for “DeviceCommunicator.exe”.

Expected result: An error message is shown, including a device communicator exit code.

4.2.3 Error while unlocking the database

1. Open the database created in subsection 4.1.1.
2. In the “Open Database” dialogue, select “Key File: FIDO2 Key Provider” as unlock method and click “OK”.
3. Disconnect the authenticator from the computer.
4. Submit the “FIDO2 PIN” dialogue.
5. Confirm the UAC prompt for “DeviceCommunicator.exe”.

Expected result: An error message is shown, including a device communicator exit code.

Conclusion

As a first step, this thesis was meant to analyze the possibilities provided by the KeePass plugin system and review the implementations of existing key provider plugins. This analysis was covered in section 1.1 and could be summarized by saying that KeePass has a flexible, well-documented plugin system, which allows implementing alternative database unlock methods.

As a second step, we aimed to analyze the capabilities of FIDO2 and design a suitable approach to implementing a key provider plugin utilizing a FIDO2 device, with a focus on security and ease of use. We analyzed the FIDO2 specifications in section 1.2, specified the key design goals on a technical level in section 2.1, and then discussed four implementation approaches to achieve them in section 2.2.

We found that the first two implementation options, which would otherwise be the most suitable ones, are not possible due to the limited capabilities of FIDO2 devices. Later, we found that the third option would only work with some authenticators, which opted not to implement one of the features of the FIDO2 specifications. This left us the fourth option as the only one that should work with any device.

In chapter 3, we fulfilled the final goal, which was implementing the KeePass plugin. We have found, however, that due to restrictions on access to FIDO2 devices in the recent versions of Windows, the plugin will only work when running under a privileged account. In chapter 4, we described test cases, which can be used to verify the plugin functions correctly.

Even though the work meets all initial goals and was done in the “best possible” way, there were many technical obstacles that forced us to use subpar solutions to some problems, for the lack of better options. For that reason, we consider the most notable result of this work to be the detailed analysis of the current options.

Future versions of FIDO specifications or new versions of the native Windows API may solve some of the current issues and allow a better implementation approach.

Bibliography

[1] Reichl, D. *KeePass Password Safe*. [online], 2003–2020, [accessed 2020-05-09]. Available from: https://keepass.info/

[2] Reichl, D. *Composite Master Key – KeePass*. [online], 2003–2020, [accessed 2020-05-09]. Available from: https://keepass.info/help/base/keys.html

[3] Reichl, D. *Installation/Portability – KeePass*. [online], 2003–2020, [accessed 2020-05-09]. Available from: https://keepass.info/help/v2/setup.html

[4] Reichl, D. *Downloads – KeePass*. [online], 2003–2020, [accessed 2020-05-09]. Available from: https://keepass.info/download.html

[5] Reichl, D. *Plugins – KeePass*. [online], 2003–2020, [accessed 2020-05-09]. Available from: https://keepass.info/plugins.html

[6] Reichl, D. *Plugin Development – KeePass*. [online], 2003–2020, [accessed 2020-05-09]. Available from: https://keepass.info/help/v2_dev/plg_index.html

[7] Reichl, D. *Key Provider Development – KeePass*. [online], 2003–2020, [accessed 2020-05-10]. Available from: https://keepass.info/help/v2_dev/plg_keyprov.html

[8] Gamma, E.; Helm, R.; et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, November 1994, ISBN 0-201-63361-2.

[9] Heitzmann, D. *Multi Cert Key Provider*. [software], [accessed 2020-05-18]. Available from: https://www.creative-webdesign.de/en/software/keepass-plugins/multi-cert-keyprovider.html

[10] Heitzmann, D. *RSA Cert Key Provider*. [software], [accessed 2020-05-18]. Available from: https://www.creative-webdesign.de/en/software/keepass-plugins/rsa-cert-keyprovider.html

[11] Buchler, M. *CertKeyProvider*. [software], [accessed 2020-05-18]. Available from: https://github.com/markbott/CertKeyProvider

[12] Estelmann, J. *KeePassQuickUnlock*. [software], [accessed 2020-05-18]. Available from: https://github.com/JanisEst/KeePassQuickUnlock

[13] Sitnikov, S.; Osipkov, A. *KeePassWinHello*. [software], [accessed 2020-05-18]. Available from: https://github.com/sirAndros/KeePassWinHello

[14] FIDO Alliance. *FIDO2: WebAuthn & CTAP*. [online], [accessed 2020-05-10]. Available from: https://fidoalliance.org/fido2/

[15] W3C. *Web Authentication: An API for accessing Public Key Credentials Level 1*. March 2019, [accessed 2020-05-11]. Available from: https://www.w3.org/TR/2019/REC-webauthn-1-20190304/

[16] Yubico. *WebAuthn Introduction*. [online], [accessed 2020-05-11]. Available from: https://developers.yubico.com/WebAuthn/

[17] Yubico. *U2F – Protocol details – Key generation*. [online], [accessed 2020-05-24]. Available from: https://developers.yubico.com/U2F/Protocol_details/Key_generation.html

[18] Yubico. *Works with YubiKey Catalog*. [online], [accessed 2020-05-24]. Available from: https://www.yubico.com/works-with-yubikey/catalog/#protocol=universal-2nd-factor-(u2f)

[19] Yubico. *FIDO U2F*. [online], [accessed 2020-05-24]. Available from: https://www.yubico.com/authentication-standards/fido-u2f/

[20] FIDO Alliance. *Client to Authenticator Protocol (CTAP)*. January 2019, [accessed 2020-05-11]. Available from: https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html

[21] Reichl, D. *KDBX 4 – KeePass*. [online], 2003–2020, [accessed 2020-05-20]. Available from: https://keepass.info/help/kb/kdbx_4.html

[22] Callas, J.; Donnerhacke, L.; et al. *OpenPGP Message Format*. RFC 4880, RFC Editor, November 2007. Available from: https://www.rfc-editor.org/rfc/rfc4880.txt

[23] IANA. *IANA CBOR Object Signing and Encryption (COSE) Algorithms Registry*. [online], [accessed 2020-05-21]. Available from: https://www.iana.org/assignments/cose/cose.xhtml#algorithms

[24] Moriarty, K.; Kaliski, B.; et al. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017, RFC Editor, November 2016.

[25] Goldreich, O. *The Foundations of Cryptography - Volume II, Basic Applications*. New York: Cambridge University Press, May 2004, ISBN 0-521-83084-2, 498 pp.

[26] Internet2. *EduPerson Object Class Specification*. [online], [accessed 2020-05-25]. Available from: https://www.internet2.edu/media/medialibrary/2013/09/04/internet2-mace-dir-eduperson-200604.html

[27] Masinter, L. *The "data" URL scheme*. RFC 2397, RFC Editor, August 1998. Available from: https://www.rfc-editor.org/rfc/rfc2397.txt

[28] Yubico. *libfido2*. [software], [accessed 2020-05-24]. Available from: https://github.com/Yubico/libfido2

[29] Corporation, M. *libfido2*. [online], [accessed 2020-05-24]. Available from: https://github.com/microsoft/webauthn

[30] Saltzer, J. H.; Schroeder, M. D. The protection of information in computer systems. *Proceedings of the IEEE*, volume 63, no. 9, 1975: pp. 1278–1308.

[31] Wenzel, M.; et al. *Consuming Unmanaged DLL Functions*. [online], [accessed 2020-05-25]. Available from: https://docs.microsoft.com/en-us/dotnet/framework/interop/consuming-unmanaged-dll-functions

Appendix A

Acronyms

AES Advanced Encryption Standard

API Application programming interface

CBOR Concise Binary Object Representation

COSE CBOR Object Signing and Encryption

CTAP Client-to-Authenticator Protocol

DLL Dynamic-link library

DPAPI Windows Data Protection API

GPL General Public License

RP relying party

TPM Trusted Platform Module

U2F Universal 2nd Factor

UAC User Account Control

W3C World Wide Web Consortium

WebAuthn Web Authentication

Appendix B

Glossary

authenticator

A cryptographic entity that handles generating and storing keys, and performing cryptographic operations.

challenge

A randomly generated piece of data that the authenticator is expected to sign.

client

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

client device

A hardware device on which the client runs, e.g., a smartphone or a laptop.

database

Generally, an organized collection of data. In this work, database refers specifically to the file in which KeePass stores usernames, passwords, and other associated data.

FIDO Alliance

An open industry association focused on authentication standards that aim to reduce the use of passwords. Members include many large technology companies, such as Amazon, Apple, Google, or Microsoft.

key source

A master password, key file, or other secret data.

master key

An encryption key based on one or more key sources.

OpenPGP

An email encryption standard defined by the OpenPGP Working Group of the Internet Engineering Task Force (IETF) as a Proposed Standard in RFC 4880.

platform authenticator

An authenticator that is attached using a client device-specific transport and is usually not removable.

PLGX

An optional plugin file format for KeePass $\geq$ 2.09. Instead of compiling the plugin to a DLL file, the plugin source code files are packed into a PLGX file and KeePass compiles them itself when the plugin is first loaded [6].

relying party

An entity whose application utilizes the WebAuthn API.

resident credential

A credential whose private key is stored in the authenticator, client, or client device.

roaming authenticator

A roaming authenticator is attached using cross-platform transports, removable, and can "roam" among client devices.

TPM

A secure crypto-processor that is designed to carry out cryptographic operations and store cryptographic keys.

user verification

A process by which the authenticator locally authorizes the invocation of its operations. User verification may be instigated through, for example, a touch plus pin code, password entry, or biometric recognition.

Windows Credential Manager

A place where Windows and other apps using its API store credentials scoped to a specific Windows account.

Windows Hello

A technology that adds alternative way to authenticate into Windows and applications using a fingerprint, iris scan, facial recognition, a short PIN, or other method.

Appendix C

Contents of the enclosed SD card

```
README.md ........................ contents description
KeePassFIDO2-v1.0.0.zip ........... plugin binaries
DeviceCommunicator ........ source code of the communicator module
KeePassPlugin ............. source code of the plugin
Thesis .................... LaTeX source code of the thesis
  BP_Kolárik_Martin_2020.pdf ....... PDF version of the thesis
```