POLITECNICO DI TORINO

Department of control and computer engineering

Master Degree Thesis
in
Computer Engineering

A Secure Password Wallet based on the SEcube™ framework

Supervisors
Prof. Paolo Ernesto Prinetto
Dr. Giuseppe Airò Farulla

Candidates
Walter Gallego Gómez
matricola: s225140

July 2018

To my mom
† In memory of my father

Summary

Nowadays, having a large quantity of digital passwords is the norm, and as their number increases, it becomes impossible to memorize all of them. This is specially true considering one should adopt strong passwords which are in general very long and complex, and therefore hard to remember. This has led users to rely on software applications to manage their passwords, the most common cases being web browsers and password wallets. The drawback of this approach is that security may be compromised, since all the passwords are stored in the same place and an attacker could gain access to them. For users particularly interested in ensuring the security of their systems, this software based approach may not be acceptable. This work presents an alternative solution based on the SEcube™ framework that guarantees the security of the stored passwords.

The SEcube™ (Secure Environment cube) framework consist of an open source security-oriented hardware platform designed by the Blu5 Group, and a set of open source software libraries developed by European research institutions. The core of the framework is the SEcube™ chip, which integrates three key security elements in a single package: A fast floating-point Cortex-M4 CPU, a high-performance FPGA and an EAL5+ certified Security Controller (Smart Card). These elements, in conjunction with a set of custom software libraries allow developers to implement highly reliable security applications.

The desktop application developed in this work, named **SEcubeWallet**, was written in C/C++ and Qt. It manages passwords using SecureSQlite, one of the SEcube™ libraries, which wraps the functionalities of the SQlite standard to create SEcube™ secured databases. In short, the data of interest is encrypted using the SEcube™ device, and can only be decrypted if the device is connected and the user authenticates using a master password. As the core operations are performed by the device, not by the host, the encryption/decryption can be done in any computer where an appropriate version of Qt is installed and the device is connected.

As front end, the application presents the user with a pleasant and intuitive

graphical user interface. With it, the user can easily create, delete, open, and modify password wallets. The GUI is easily configurable and is cross-platform. Additionally, the application can suggest strong Passwords and Passphrases and verify the entropy of the ones provided by the user.

In conclusion, SEcubeWallet is an excellent application for the storing and management of passwords. As it relies on the SEcube™ framework it is trustworthy and the information is virtually impossible to steal. Thanks to the developed GUI it is easy to use, and it offers some interesting functionalities to increase the user experience.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This thesis work regards the design and development of a hardware-based password manager system. To justify its relevance inside the current digital security ecosystem, three key questions need to be addressed.

- Are passwords still relevant?
- Why should people use password managers?
- Why are hardware-based approaches more reliable?

The answer to the first question is pretty straightforward. Yes. Passwords are, to the date, the dominant form of authentication in a lot of scenarios, including computer/server logins and web services. Even if other forms of authentication are already being widely use (hardware token devices and one-time passwords in banking; biometrics in smartphones), those applications still depend on passwords either as a fallback system (a smartphone will ask the user for a pin/password if the fingerprint recognition failed) or as a complementary security measure (to generate a one-time password, the user must first enter a regular password).

Unauthorized access to computers or smartphones, a web service, banking information or company servers, all can have catastrophic consequences for victims. Personal data such as photos and emails, intellectual properties, money and even somebody's identity are just a few examples of what authentication systems are protecting, reasons more than enough to be concerned about the reliability of passwords.

Because passwords use is omnipresent, one would expect it to be a highly secure authentication method. This however, is in general not true. Since people have a large an increasing number of passwords that have to be memorized, they tend to use ones that are not strong but rather, easy to remember. The three most common bad practices are:

- Using short and low complex passwords that include common sequences or words.
- Using passwords with some significance, like a birthday or a pet's name.
- Reusing the same password for multiple services, with small modifications or non at all.

With password managers the story is different. People no longer need to worry about remembering their passwords, managers will do that for them. Usually they also offer the capability to generate long, complex and completely random passwords. As a result, people can use a strong and unique password for each of their accounts, thus making attackers job infinitely harder.

The use of a password manager seems like the perfect solution, but there is one concern that arises. If all of the passwords are stored in the same place, it will become for sure a new target for attacks. Therefore the manager needs to be as robust and trustworthy as possible. Software-based managers usually work with a master password to encrypt/decrypt the data (i.e the other passwords). This means an attack could be carried out either by:

- Cracking the encryption algorithm used by the manager
- Cracking the user's master password. This usually happens if an attacker has access to the encrypted data and have the ability to try billions and billions of passwords until they guess the right one.
- Corrupting the manager application or the host machine OS.

The algorithms used by good password managers are usually standard ones, meaning they are the state of the art, and therefore sturdy. The weak points of the system may be in the master password and in the application being corrupted. A hardware-based manager boost the security of the system by improving in this two points.

A hardware-based manager uses a two-factor authentication method. In order to encrypt/decrypt the data, two elements are required: a master password and a portable and unique device which is connected to the host machine (user's computer for instance). Therefore, even if an attacker has access

to the encrypted data, without the device, they can not even start trying to crack the master password.

Regarding the second point, in a lot of cases the portable device is the one doing all the actual encryption/decryption of data. The host machine is only used to provide the GUI so the user can enter their master password and to display their protected passwords. As the portable device is custom designed to be as secure as possible, it is much more harder to corrupt than an OS or a software application.

In conclusion, storing and protecting passwords is a major goal in digital security, and one of the best approaches to the date are hardware-based managers. This work regards one of them, implemented as a desktop application that exploits the capabilities of the SEcube™ (Secure Environment cube) hardware and software framework. The core of the framework is the SEcube™ chip developed by the Blu5 Group[4], which integrates three key security elements in a single package: A fast floating-point Cortex-M4 CPU, a high-performance FPGA and an EAL5+ certified Security Controller (Smart Card). This chip, in conjunction with a set of custom device-side software libraries[14] developed by European research institutions, act as the password manager's hardware device, and is in charge of authenticating the user and encrypting/decrypting the data.

The desktop application, named SEcubeWallet, was written in C/C++ and Qt, and it interacts with the SEcube™ device, requesting services like authentication and encryption. Its main tasks are:

- Manage the set of passwords (hereinafter referred to as a Wallet) using secureSQLite, one of the SEcube™ host-side libraries, that works by wrapping the functionalities of the SQlite standard to create SEcube™ secured databases.
- Serve as GUI so the user can authenticate to the SEcube™ device and create, open, edit, save and delete wallets with ease. The GUI displays the wallet's content in a table view and each column can be filtered individually.
- Suggest strong Passwords (and Passphrases) which can be used with confidence in any login service. The application also verifies the entropy (strength measure) of the generated passwords, or of the ones provided by the user.

The remaining of the thesis is organized as follows:

Chapter 2 gives an overview of some existent hardware-based password managers and of other applications based on the SEcube™ framework.

In Chapter 3 the application development process is explained, by covering its general design, the hardware and software libraries used, and the actual implementation showing also relevant portions of code.

Chapter

Chapter 2

Related Work

This chapter gives a review of a few projects that are related to this work. The first section deals with projects that have as objective the securing of passwords using a hardware approach. The second section regards applications using the SEcube™ framework.

2.1 Hardware Password Wallets

Over the last decade, ...

2.1.1 Mooltipass: A Simple Offline Password Keeper

The Mooltipass is a hardware-based password keeper commercial product that allows users to safely carry their credentials with them all the time, and use them when needed by connecting a portable device to a computer and authenticating using a 4 characters pin. Figure 2.1 depicts the mooltipass system.

Its use is very, simple, as explained in their official website [9]:

"The Mooltipass is designed to be as simple as possible to use for users of all backgrounds and ages:

1. Plug the Mooltipass to your computer/tablet/phone. No driver is required
2. Insert your smartcard, unlock it with your PIN. Without the PIN, the card is useless.
3. Visit a website that needs a login. If using our browser plugin, the Mooltipass asks your permission to send the stored credentials, or asks

Figure 2.1: The mooltipass device and smartcard

you to save new ones if you are logging in for the first time.

4. If you are not using the browser plugin or are logging in on something other than a browser, you can tell the Mooltipass to type your logins and passwords for you, just like a keyboard.

The Mooltipass emulates a standard USB keyboard, and can therefore type your passwords for you on Windows, Linux, Mac and even most Apple and Android devices (through the USB On-The-Go port). It doesn't need any special drivers to function. "

The browser plugins offer a GUI where the user see and manage their passwords, as shown in Figure 2.2

The Mooltipass works by storing an encrypted version of the passwords, and allowing their decryption only when the proper smartcard is connected and the correct pin introduced. From the website:

"The Mooltipass has an internal flash in which the user encrypted credentials are stored, while a PIN-locked smartcard contains the AES-256bits key required for their decryption. Like any chip and pin card, 3 false tries will permanently disable the Mooltipass card. Credentials are sent over HID, any password accessing operation needs to be physically approved by the user on the device."

Mooltipass is open software and open hardware. In fact they encourage the community to develop and review code, so that the system's reliability is increased. Their GitHub repository host all the sources from the beginning of the project. [10].

The Mooltipass project started off as a Hackday post from developer Mathieu Stephan, that gained enough recognition by the community to be founded

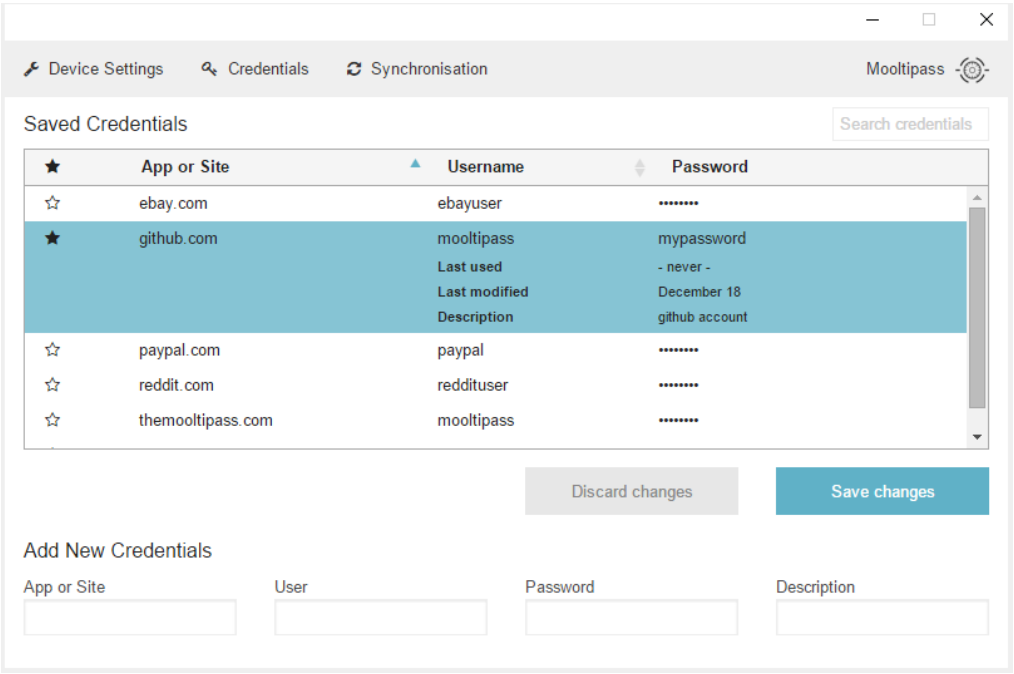


Figure 2.2: The mooltipass application

by a kickstarter campaign. The Hackday project website [11] contains more details regarding the hardware implementation:

- **ST662ACD-TR:** Power Management
- **ATMEGA32U4-MU:** Arduino compatible Microprocessor
- **AT88SC102:** Secure Memory Smart Card
- **AT45DB011D-SSH-T:** FLASH Memory

Figure 2.3 shows the basic architecture.

In conclusion, mooltipass is a mature project that follows the same idea developed in this work: Guarantee the security of a set of passwords by allowing their decryption only through the use of a hardware device that the user will carry with them, hopefully, all the time. However there are some important difference between their system and SEcubeWallet:

1. SEcubeWallet is based on the SEcube™ platform, making it much more trustworthy and robust, as this is a platform even used in military applications, and has been tested in the most demanding conditions.
2. Because SEcube™ integrates all the required security elements into one single chip, the final product is much more smaller (The size of a regular USB stick) than the mooltipass device.

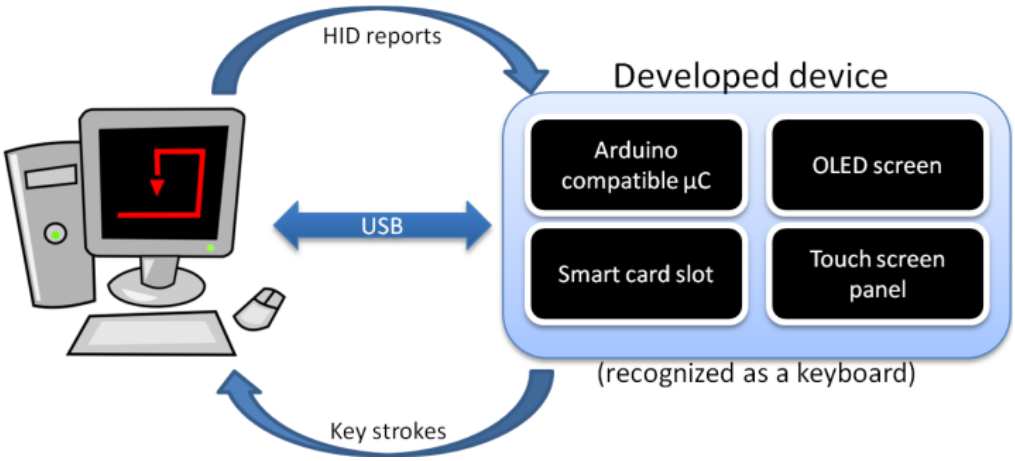


Figure 2.3: The Mooltipass basic architecture

3. Because the SEcube™ platform offers more possibilities for the development of applications, for example using the open source libraries, or the FPGA inside the chip, the SEcubeWallet can be extended or be integrated with other projects.
4. SEcubeWallet offers a couple of additional functionalities: Strong Password generation and entropy estimation, increasing the user experience.

2.2 SEcube™ based applications

Before starting the application development, the following three projects were studied in order to familiarize with the SEcube™ framework and libraries usage. All of the projects make part of the SEfile SDK available at [14] and are covered in details in the L2 user Manual [21].

2.2.1 Secure Text Editor and Secure Image Viewer

This two projects are very similar, both use the SEcube™ framework, in particular the `SEfile` library, to encrypt/decrypt plain text files and images respectively. They are not intended to be final user applications, but demos that allow developers to learn how to use the SEcube™ libraries.

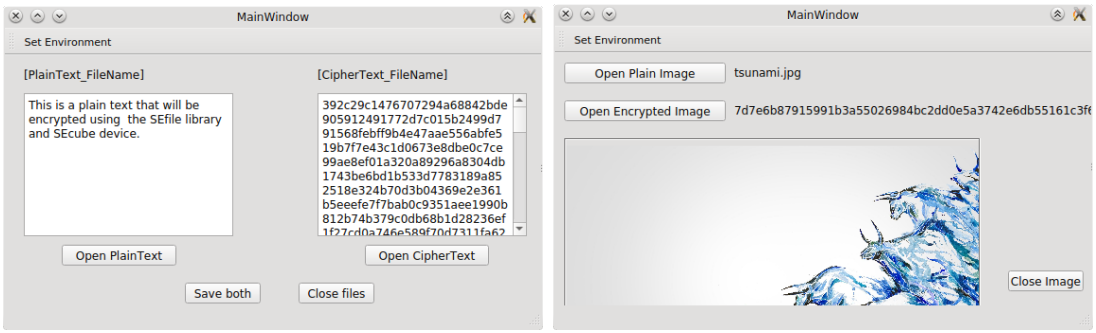
"Both these two projects have been developed in C++ with Qt libraries. They are based on 3 major security classes, in a one-to-one mapping with the 3 most important security opera- tions: the first one manages the security platform to which the user wants to log in, the sec- ond one allows the

selection of the secure environment through the `secure_update()` function, while the third one manages the opening and creation of files resorting on the `secure_ls()`."[21].

The secure text editor, in short `SEfile_TXT` offers the possibility to create/edit/open plain text documents, and generate an encrypted version of the file that will be stored in the same directory. "It is possible to verify that encrypted files cannot be read properly from regular text editors; conversely, the Secure Text Editor can transparently read any encrypted file (decrypting also the file name) which content has not been altered and is, thus, trusted. Unauthenticated content (i.e., content not corresponding to the file signature) is, instead, discarded".

The secure image viewer, in short `SEfile_IMG` allows the user to open the most common image formats, PNG JPG/JPEG and BMP, and generates the encrypted version, which can only be opened using the same application.

Figure 2.4 depicts these two demo applications.



(a) SEfile_TXT demo application (b) SEfile_IMG

Figure 2.4: SEfile demo applications

These two demo applications were used to learn the basis of the SEcube™ libraries, specially how to open the communication with the device, how to authenticate (login) and how to logout. The login dialogue used in the SEcubeWallet application is an improved version of these demos' login dialogue. The environmental dialogue was borrowed without any modification.

2.2.2 secureSQLiteBrowser

This application integrates the SEcube™ secureSQLite library with the DB Browser for SQLite[6] project, resulting in a powerful manager of encrypted SQLite databases.

The secureSQLite library, that makes part of the SEfile SDK, modifies the SQLite system to use the SEfile library in order to manage files, rather than using directly the OS calls. The result is a library that accepts all the standard SQLite commands, but stores the database as an encrypted file.

DB Browser for SQLite is an application developed in Qt that allows to manage SQLite DB from a powerful GUI. "DB Browser for SQLite is a high quality, visual, open source tool to create, design, and edit database files compatible with SQLite. It is for users and developers wanting to create databases, search, and edit data. It uses a familiar spreadsheet-like interface, and you don't need to learn complicated SQL commands." [6]

By merging this two projects, the result is a wonderful application to create encrypted SQLite databases using an elegant and powerful GUI, depicted in figure 2.5, with tons of options, where the users can visually edit the DB tables, and store them as SEcube™ secured files.

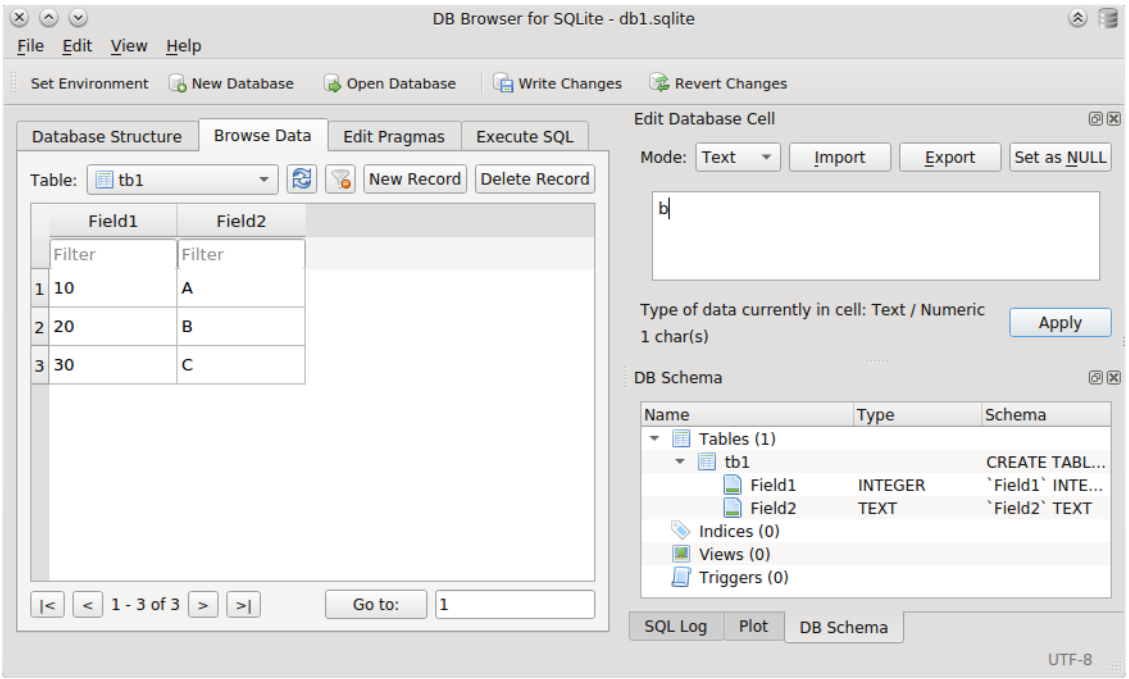


Figure 2.5: secureSQLiteBrowser GUI

Because SEcubeWallet is heavily based on the use of the secureSQLite library, this application was used to learn how to integrate the library into a Qt project. Additionally, it was used as inspiration for some of the GUI elements, like the filters above each column and the use of menus and toolbars. It was also use to diagnose the SEcubeWallet in the developing stage,

by creating a DB with one application and opening it in the other. secureSQLiteBrowser could be use as a password manager in the sense that it can securely store and display wallets as tables, but it lacks the simplicity and additional functionalities that make SEcubeWallet attractive to users.

Chapter 3

Application Development

3.1 Useful concepts definition

More detailed explanations of the following concepts will be given later on, but it is helpful to shortly define them here so this work is more readable.

3.1.1 Wallet

A password wallet is a digital form of securely keeping passwords and some meta information. In this work, a Wallet is stored as a SQLite DataBase. A wallet can have as many tables as the user wants. For instance an user could have a table for storing social media passwords, another one for work-related passwords and a last one for credit cards and bank accounts passwords. Finally, each table has a set of defined fields (Username, Domain, Password, Date, Description).

3.1.2 SEcube

SEcube™ is a custom chip produced by the Blu5 group [4] that integrates an ARM CPU, a FPGA and a SmartCard. The chip is specifically designed for security purposes, allowing developers to implement encryption/decryption functions that are executed fast and are guaranteed to be reliable. The chip can be connected to a PC using popular protocols like USB and Ethernet, so an application running on the PC can use the SEcube™ to encrypt/decrypt some data.

3.1.3 SEcube™ SDK

The SEcube™ Open SDK is a set of open libraries designed to make the development of applications using SEcube™ more convenient. There are two types of Libraries: Host side (PC) and device side (SEcube) libraries. In general host side functions make requests to device side functions and wait for their response. Moreover, libraries are divided in four and two hierarchical levels of abstraction, for host and device side respectively. Level0 and Level1 are the lowest levels and are present in both host and device. L0 provides the communication protocols while L1 provides basic security APIs.

3.1.4 SEfile

SEfile is a Level 2 API that allows users to encrypt/decrypt data (files in the hard disk), so they can only be read when the SEcube™ chip is connected to the PC. When the SEcube™ is not connected, it is impossible to read the files as the information necessary to decrypt them is physically stored in the SEcube.

3.1.5 SQLite DB

3.2 Design

The purpose of this section is to give the reader a clear overview of how the application works in general terms.

A simplified design architecture is displayed in figure 3.1. It shows which Software Libraries are used by the application and when it uses them.

The following is a brief explanation of these Libraries and they usage. More details about each Library and why they were chosen are given in section 3.3 and section 3.4 deals with the actual implementation.

3.2.1 L0 and L1 Authentication libraries

When the user starts any application based on the SEcube™ platform, the first steps to perform are to open the communication with the device using Level0 functions from both host and device, and to authenticate the user by checking the login pin, using Level1 functions (again, from both sides). Figure 3.1 depicts how the SEcubeWallet uses the authentication functions and they in turn communicate with the SEcube™ chip.

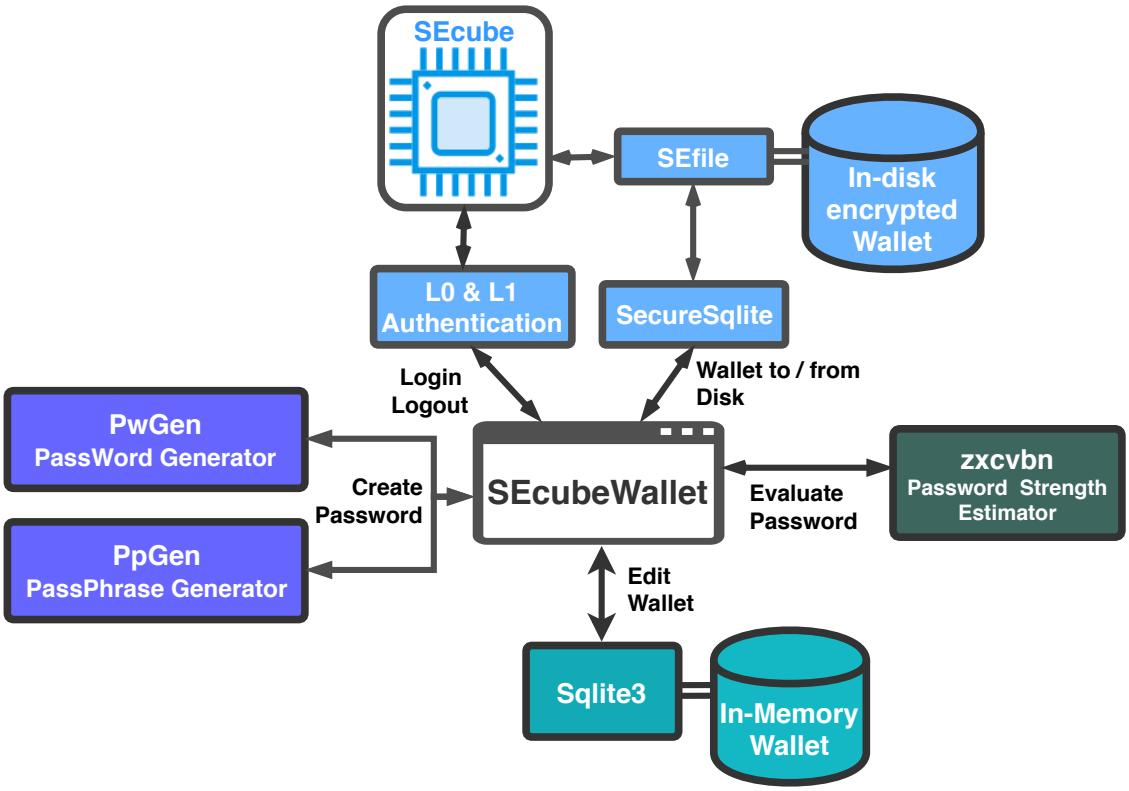


Figure 3.1: Basic Design: Used Libraries

3.2.2 secureSQLite3

As explained before, Wallets are stored as SQLite DataBases. Fortunately, the SEcube™ SDK already provides a Level2 API for creating and managing encrypted SQLite DBs, called secureSQLite3. This API exploits the SEfile API to wrap some of the functions of the original SQLite3 library, avoiding OS calls.

secureSQLite allows to create/edit/save/open databases that when written to the disk are encrypted and can only be read when the SEcube™ is connected. Additionally, as it is implemented using wrappers, the developer only needs to include the source files in the project and can manage secureSQLite DB with the same functions used for regular SQLite DB.

Figure 3.1 shows the SEcubeWallet application using secureSQLite to read/write the encrypted wallet stored in disk.

3.2.3 SQLite3

Because the use of secureSQLite involves a call to the SEcube™, a regular SQLite3 DB is also used, but unencrypted data is never saved to the disk. SQLite3 allows for the creation of an In-Memory DB, i.e. a DB whose content is always in the application's memory space and is therefore secured by the operating system.

The In-memory DB is used for editing. When the user want to save the wallet, i.e. write it to the disk, the contents of the In-Memory DB are dumped to the encrypted secureSQLite DB. When the user opens a wallet from the disk, the reverse process occurs.

With the In-memory DB, unnecessary calls to the SEcube™ are avoided while maintaining the contents secured.

3.2.4 Password Generator

As the purpose of the application is to securely store passwords, said passwords should be as strong as possible. It does not make sense to protect a password that can be easily cracked by a hacker using brute force. That is why the application also includes a Password Generator.

PwGen [26] is an open source library that generates passwords, that can either be easy to remember, or completely random. Random passwords are more secure, but as the are difficult to remember, their use only makes sense when the user stores them in a wallet manager. Among other aspects, length and characters used (Numbers, Upper cases) can be configured too.

When the user is adding a new entry to a wallet, they can chose to enter a password or to automatically generate one.

3.2.5 PassPhrase Generator

In addition to the Password Generator PwGen, the user has the possibility to generate PassPhrases instead. PassPhrases are a popular alternative because they are easier to memorize and therefore can be longer, which in turns make them more secure. Further details about the usefulness of PassPhrases and how they compare against regular Passwords is given in section 3.3.5.

Although the PassPhrase Generator is not a library per se, it was included in the diagram because of its close relation with the PwGen and zxcvbn [17] libraries. It was developed by the author, and it works by selecting random words out of dictionary files.

The user can configure how many words each passphrase must have, the minimum length of said words, and which dictionary files to use, among other options.

3.2.6 Strength Estimator

To give the users feedback on how good the password they are about to store is, the application uses the open source project zxcvbn to give an estimation of the passwords entropy, and how long it would take for a hacker to break it. zxcvbn evaluates if the password is a common word, or a combination of them, a last name, a date, or letters close to each other in a keyboard (thus the name zxcvbn). With the estimator users are encouraged to create good passwords that are not necessarily completely random and difficult to remember, or annoying to type.

To recap, these are the key aspects of the used libraries:

- To start the connection with the SEcube™, the Level0 library is used
- To authenticate the user, by checking if the entered login pin is the same as the pin stored in the SEcube™, the Level1 library is used.
- An in-memory database is used for editing the wallet.
- An encrypted in-disk data base is used for storing the wallet in disk.
- The application includes a password generator with several options.
- The application also includes a password strength estimator, so the user has an idea of how good their passwords are.

3.3 Frameworks, Libraries and software tools

As explained in the previous section, the core of the design is the use of the SEcube™ chip to perform security operations in order to encrypt/decrypt some data stored in the host (PC). The requests to the chip are made from the Qt application developed in this work, which runs in the host. Said application exploits the existing C libraries SEfile and secureSQLite to ease the communication with the SEcube™. Additionally, the application also makes use of a random password generator PwGen and a strength estimator zxcvbn open libraries.

In the following sections a review of the SEcube™ platform's hardware and software components is given. Then a brief explanation of the C++/Qt framework and why it was chosen. Finally the additional used libraries are presented.

3.3.1 The SEcube™ framework

"The SEcube™ (Secure Environment cube) Open Security Platform is an open source security oriented hardware and software platform, designed and constructed with ease of integration and service-orientation in mind. The hardware part of the platform was originally designed by Blu5 Group [4], whereas the software libraries stem from a strong cooperation among international research institutions." [20].

The main **hardware** products, explained in detail in the following sections, are:

- The Chip, named SEcube™ Chip, or simply **SEcube™**
- The Development Board, named **SEcube™ DevKit**
- The USB Stick, named **USEcube Stick**.

The SEcube™ chip is the main hardware component, and both the devkit and USB Stick are designed around it.The Development Board provides several communication protocols as well as debugging capabilities. For the final product the board would be of course too inconvenient to carry, and instead the USEcube Stick is preferred.

The SEcube™ Chip

"The SEcube™ (Secure Environment cube) is a powerful chip which integrates three key security elements in a single package. A fast floating-point Cortex-M4 **CPU**, a high-performance **FPGA** and an EAL5+ certified Security Controller (**Smart Card**). The result of this innovative combination gives an extremely versatile secure environment in a single SoC, in which developers can rapidly implement complex applications and appliances. ... The SEcube™ is the ultimate solution for high-end design, delivering integration of a flexible, configurable and certified secure element." [19]

We can then see the SEcube™ chip as a powerful device offering the flexibility of an ARM CPU, the speed of an FPGA and the reliable security of a certified Smart Card, all bounded together and easily integrated in any

project thanks to the available communication protocols, among them USB, UART, Ethernet and JTAG.

The chip includes a true random number generator which relies in 240 noise seeds, all physical and therefore unpredictable. This allows the creation of true random noise. Additionally the user can choose what type of noise they want to generate, for instance white or Fourier noise.

In figure 3.2 a simplified SEcube™ architecture is shown.

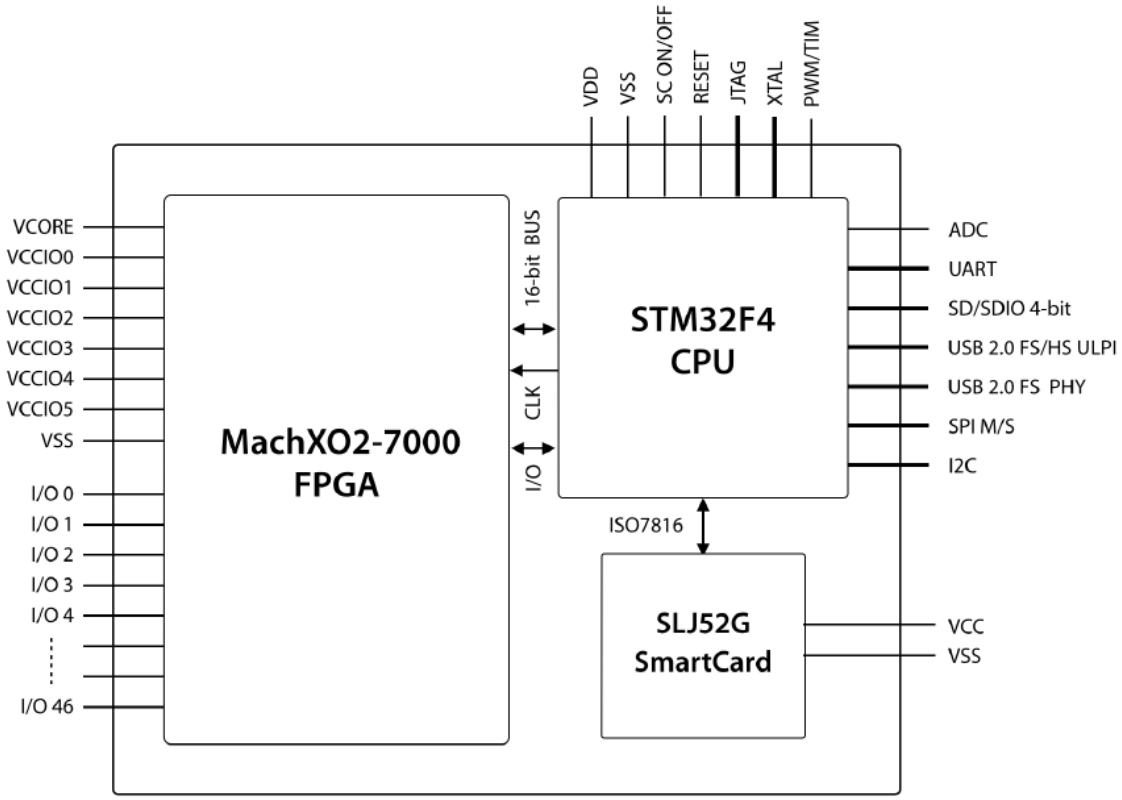


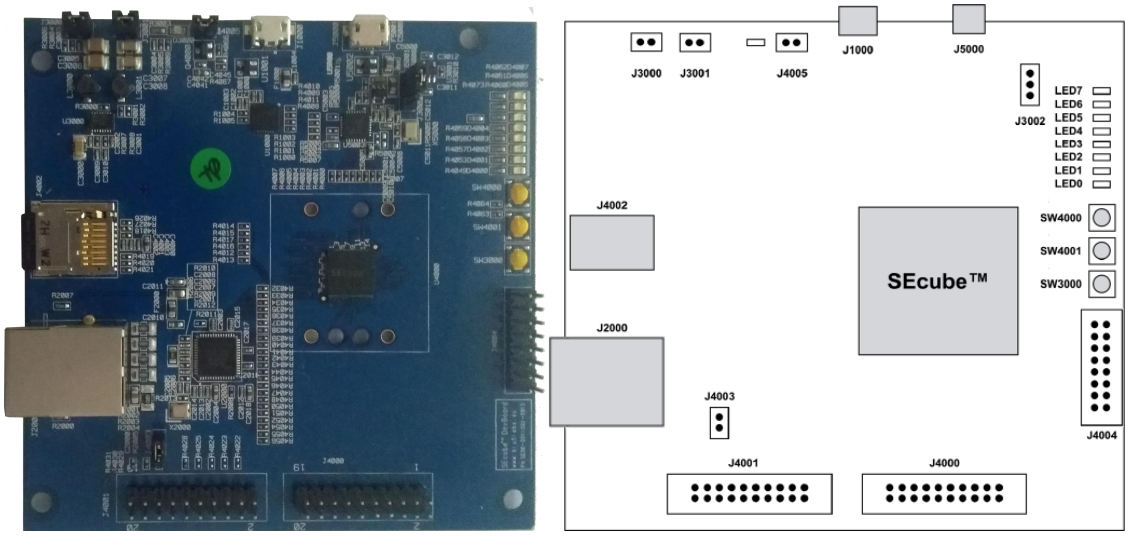
Figure 3.2: SEcube™ Block Diagram

Development board: The SEcube™ DevKit

The development board integrates the SEcube™ chip with several peripherals that allow the user to easily communicate, program and debug. (Figure 3.3)

The main peripherals in the SEcube™ devkit are:

- **J1000:** USB 2.0 to UART
- **J2000:** Ethernet 10/100 socket



(a) (b)

Figure 3.3: SEcube™ Devkit

- **J4000:** SEcube™ embedded FPGA and CPU GPIOs
- **J4001:** SEcube™ embedded CPU JTAG
- **J4002:** microSD card
- **J4004:** SEcube™ embedded FPGA and CPU GPIOs
- **J5000:** USB 2.0 High Speed
- **LEDx:** Leds
- **SWx00y:** Switches

Final product: USEcube Stick

For the final product, its is desired that the user carries all the SEcube™ functionalities in a small and convenient package, so they can encrypt/decrypt the passwords in any PC by just connecting the USEcube Stick and running the SEcubeWallet application.

The USEcube Stick is compatible with any Operating System and the SEcube™ functionalities are easily exposed to applications and services without installing any driver.

The USEcube offers only the strictly required components: The SEcube™ chip, a USB 2.0 High-Speed interface and an SDcard socket. See Figure 3.4 for more details.

Since the USEcube Stick storage capability is based on a external microSD card, the security of the system is improved, as this allows to have a separation of encrypted data from the encryptor/decryptor. Additionally, both the size and the speed can be tuned per the user requirement and can be changed at any time, just replacing the microSD, without buying a new USEcube Stick. The microSD card socket is embedded in the USB connector allowing to save space making the USEcube Stick very compact and, at the same time dust and water-resistant. Since the USEcube Stick is not provided with the JTAG interface, to inject the firmware previously developed and tested on the SEcube™ DevKit, all the devices come with an embedded secure boot loader.

(a) (b)

Figure 3.4: USEcube Stick

L2 Security APIs

"The software libraries and design environment allow developers who are not willing or able to produce the security APIs and protocols themselves to exploit the ready functions provided (currently as APIs and soon as services) within the SEcube™ platform and experience the platform as a high-security black box." [21]

"From the user/developer point of view, the APIs have been implemented targeting two nested environments depending on where physically the code runs:

- **Device-Side**, including the libraries of basic functionalities that are

executed on the embedded processor of the SEcube™-based hardware device.

- **Host-Side**, containing libraries of functions executed on the host PC and interface functions for calling services and processes residing on the embedded processor of the SEcube™ device.

From the architectural point of view, the Host-Side Libraries have been implemented targeting 4 hierarchical abstraction levels, and namely:

- **Level 0:** Communication Protocol and Provisioning APIs
- **Level 1:** Basic Security APIs (Level1 Host-Side – L1)
- **Level 2:** Intermediate Security APIs (Level2 – L2)
- **Level 3:** Advanced Security APIs (Level3 – L3).

At each level, each component represents a "service" for the upper level and relies on "services" provided by the next lower level, only." [21]

"Level L2 relies on L1 services to provide the APIs for implementing more abstract secure functionalities. Typical examples include APIs for the protection of data both at rest and in-motion, or negotiating parameters (e.g., keys, algorithms) for establishing secure sessions, without being forced to understand in details all the low-level hardware and security mechanisms."[21]

L2 can be considered as the merge of two projects: **SEfile**, concerning data at rest, and **SElink**, concerning instead data at motion.

For our project we rely heavily on the development tools provided by the SEfile project, for the secure storage, usage and retrieve of data that requires a high degree of confidentiality, in our case, digital passwords.

SEfile

"SEfile targets any user that, by moving inside a secure environment, wants to perform basic operation on regular files. It must be pointed out that all encryption functionalities are demanded to the secure device in their entirety. In addition, SEfile does not expose to the host device details about what, or where it is reading/writing data: thus, the host OS, which might be untrusted, is totally unaware of what it is writing". [21].

secureSQLite

3.3.2 SQLite3

3.3.3 Graphical User Interface: the Qt framework

The application's graphical user interface was developed using the **Qt framework**, version 5.8.0.

"Qt is a cross-platform application development framework for desktop, embedded and mobile. Supported Platforms include Linux, OS X, Windows, VxWorks, QNX, Android, iOS, BlackBerry, Sailfish OS and others. Qt is not a programming language on its own. It is a framework written in C++. A preprocessor, the MOC (Meta-Object Compiler), is used to extend the C++ language with features like signals and slots. Before the compilation step, the MOC parses the source files written in Qt-extended C++ and generates standard compliant C++ sources from them. Thus the framework itself and applications/libraries using it can be compiled by any standard compliant C++ compiler like Clang, GCC, ICC, MinGW and MSVC".[1]

For writing, compiling and debugging source code, the IDE **Qt Creator**, version 4.2.1 was used.

"Qt Creator provides a cross-platform, complete integrated development environment (IDE) for application developers to create applications for multiple desktop, embedded, and mobile device platforms, such as Android and iOS. It is available for Linux, macOS and Windows operating systems".[12].

The reasons behind the use of Qt are as follows:

- Qt is a C++ library, and as such, allows for a seamless use of the C libraries SEfile and secureSQLite, which are the backbone of this project.
- Qt is cross-platform, meaning the developed application can be compiled to work on any of the major OSes. In particular, the development was carried out an tested on a Linux machine, but the application should work with no problems in Windows and MacOS.
- Because of good designed and ready-to-use display items such as tables, menus and dialogues, it is possible to focus in writing the functional portions of the application without worrying too much about the GUI. And as it is open source, any Qt item can be modified and extended when it does not meet the expectations out of the box. In this project several display elements were improved, as will be seen in section 3.4.

- Thanks to the multitude of functions dedicated to ease the use of C++ libraries and OS calls, one can be more productive, and the resulting code is more reliable. For instance, this project makes extensive use of such libraries, like QSqlDatabase, QString, QProcess, etc. Again, more details are given in section 3.4
- Related works by research group TESTGROUP from Politecnico di Torino using the SEcube™ framework, are written in Qt. Namely secureSQLite-Browser and SEfile_TXT, were used as base in the initial stages of development. This two projects can be found in the SEfileSDK available online [14].
- Qt is widely used, meaning it is possible to find tons of documentation, forums and additional libraries on the web. This also ensures the Qt framework will have continuous support from the developers and the community.

3.3.4 PwGen: Pronounceable Password generator

The most secure type of passwords are random ones. A random password sufficiently long is considered to be virtually unbreakable. But this rises two problems: First of all, humans are inherently bad at creating true random passwords. Second, a random password is not suited to be remembered or even used (as it probably is too annoying to type). These two reasons motivated the inclusion of a Password Generator.

pwgen is an open source program that generates human friendly passwords that are also secure. It is available in the official Linux repositories, and there is a Windows version as well, but in this work the source files where used.

"The pwgen program generates passwords which are designed to be easily memorized by humans, while being as secure as possible. Human-memorable passwords are never going to be as secure as completely completely random passwords. In particular, passwords generated by pwgen without the -s option should not be used in places where the password could be attacked via an off-line brute-force attack. On the other hand, completely randomly generated passwords have a tendency to be written down, and are subject to being compromised in that fashion" [26].

pwgen offers several options that can drastically change the type of generated password. Here is a list of the options available for users of SEcube-Wallet:

- **Length:** The desired length of the password. It is recommended to be

at least 12 for non-random passwords and 8 for random ones.

- **-0, no numerals:** Don't include numbers in the generated passwords.
- **-A, no capitalize:** Don't bother to include any capital letters in the generated passwords.
- **-B, ambiguous:** Don't use characters that could be confused by the user when printed, such as 'l' and '1', or '0' or 'O'. This reduces the number of possible passwords significantly, and as such reduces the quality of the passwords. It may be useful for users who have bad vision, but in general use of this option is not recommended.
- **-c, capitalize:** Include at least one capital letter in the password.
- **-n, numerals:** Include at least one number in the password.
- **-s, secure:** Generate completely random, hard-to-memorize passwords.
- **-v, no vowels:** Generate random passwords that do not contain vowels or numbers that might be mistaken for vowels. It provides less secure passwords to allow system administrators to not have to worry with random passwords accidentally contain offensive substrings.
- **-y, symbols:** Include at least one special character in the password.

By default pwgen behaves as if the options **-nc** were used, that is, pronounceable passwords with at least 1 capital letter and 1 number.

The strongest passwords this program can generate are obtained with the options **-ys**, as it results in random passwords with special symbols. They are very hard to remember, and should only be used if the user is willing to open the SEcubeWallet application each time they need to use one of the password.

3.3.5 zxcvbn: Password strength estimation

An important feature to have in a password manager is the possibility to realistically estimate how strong a password is, i.e., how hard could it be for hackers to crack it, as there is no point in using the SEcube™ system to protect weak passwords, that could be easily guessed with brute force attacks. As it is out of the author expertise to write a reliable function to make this estimation, it was decided to use a trusted project developed during the dropbox hackweek event in 2012. The estimator called **zxcvbn** was originally written in JavaScript aiming for an easy integration with multiple web browsers and OS. Fortunately, the community ported the library to a

wide variety of languages including Python, Ruby and C/C++. In this work the C++ implementation was used. The project is Open Source and available for free use on GitHub [17].

zxcvbn is regarded by the community as one of the most reliable and mathematically advanced open source password estimators. In security forums and discussion it always pops out as an excellent tool, much better than other passwords estimators commonly used in web pages. In [24], the author compares zxcvbn to other popular java meters and arrives to the conclusion that only zxcvbn is reliable enough to actually give an useful feedback. In [25], the author makes an evaluation of several password generators and strength estimators. PwGen and zxcvbn, the two libraries used in this work, always give excellent results.

"For over 30 years, password requirements and feedback have largely remained a product of LUDS: counts of Lower- and Uppercase letters, Digits and Symbols. LUDS remains ubiquitous despite being a conclusively burdensome and ineffective security practice. zxcvbn is an alternative password strength estimator that is small, fast, and crucially no harder than LUDS to adopt. Using leaked passwords, we compare its estimations to the best of four modern guessing attacks and show it to be accurate and conservative at low magnitudes, suitable for mitigating online attacks. We find 1.5 MB of compressed storage is sufficient to accurately estimate the best-known guessing attacks up to 105 guesses, or 104 and 103 guesses, respectively, given 245 kB and 29 kB. zxcvbn can be adopted with 4 lines of code and downloaded in seconds. It runs in milliseconds and works as-is on web, iOS and Android". [27]

"People of course choose patterns — dictionary words, spatial patterns like `qwerty`, `asdf` or `zxcvbn`, repeats like `aaaaaaa`, sequences like `abcdef` or `654321`, or some combination of the above. For passwords with uppercase letters, odds are it's the first letter that's uppercase. Numbers and symbols are often predictable as well: `l33t` speak (3 for e, 0 for o, @ or 4 for a), years, dates, zip codes, and so on. As a result, simplistic strength estimation gives bad advice. Without checking for common patterns, the practice of encouraging numbers and symbols means encouraging passwords that might only be slightly harder for a computer to crack, and yet frustratingly harder for a human to remember. xkcd nailed it". (see figure 3.5). [18]

To put it in other words, the authors of the project argue that a password like `correcthorsebatterystaple` (a nonsense English phrase) is more strong than a password like `Tr0ub4dour&3`, even if the former does

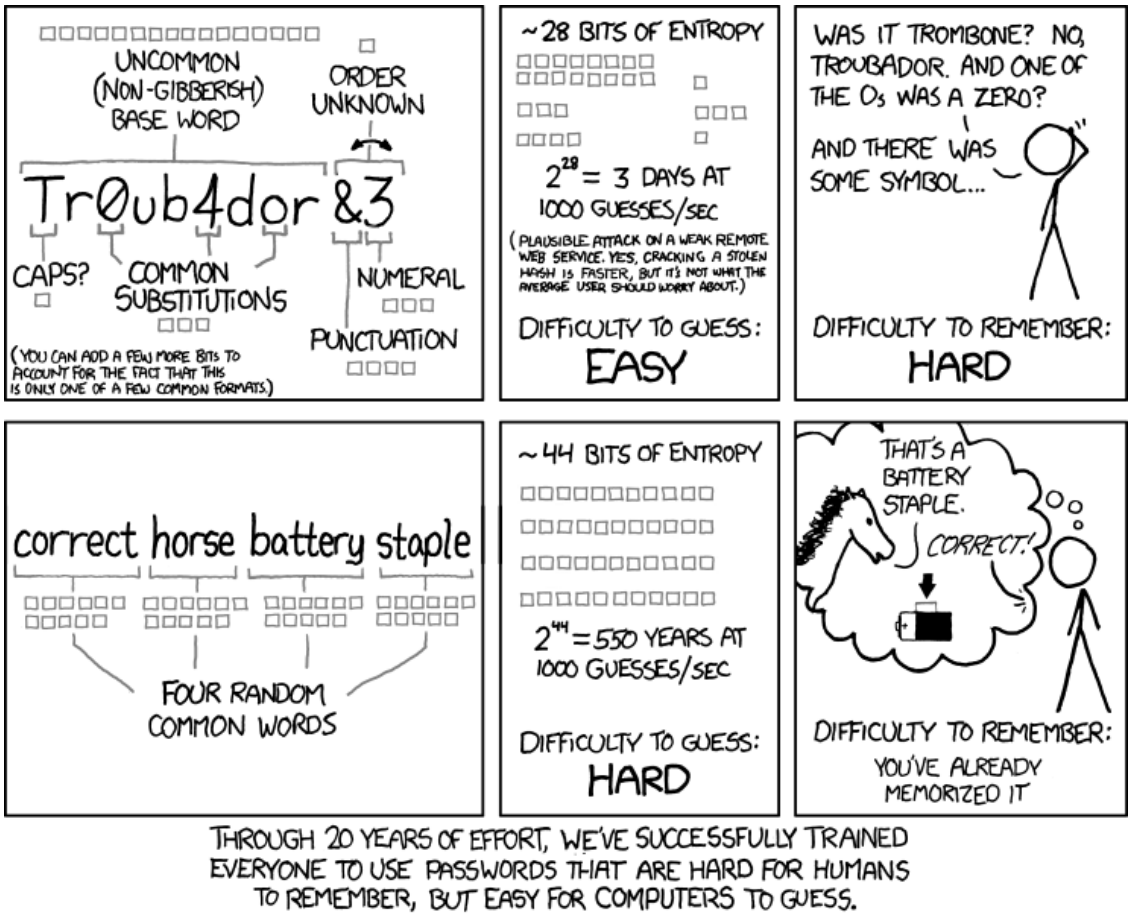


Figure 3.5: Password strength, xkcd [28]

not have any upper cases or numbers, and the latter seems more complicated.

The table in figure 3.6 (taken from [18]), show how zxcvbn is different from strength meters used in popular web services. (disclaimer: The data in the table is from 2012). From it we can learn:

1. Passwords like **`qwER43@!`**, which is a spatial password: it uses the keys `qwer4321`, with shift pressed for the keys `er` and `21` (the `@` symbol in the English keyboard is `shift+2`), is not considered week by most of the meters, but it should. It is probably due to the fact that it includes a combination of numbers and symbols that makes it look strong, but in reality, because of keyboard spatiality, is not.
2. Passwords like **`Tr0ub4dour&3`**, which is generated by replacing some of Troubadour letters with numbers, and adding two more characters, is

	qwER43@!	Tr0ub4dour&3	correcthorsebatterystaple
zxcvbn	Weak	So-so	Great!
Dropbox (old)	Great!	Great!	So-so
Citibank	Medium	Strong	1 number required
Bank of America	(not allowed)	(not allowed)	(not allowed)
Twitter	✓ Password is perfect!	✓ Password is perfect!	✓ Password is perfect!
PayPal	Weak	Strong	Weak
eBay	Strong	Strong	(not allowed)
Facebook	•••••••• Password strength: Strong	•••••••• Password strength: Strong	•••••••••••••••••••••••• Password strength: Weak
Yahoo!	Very strong	Very strong	Weak
Gmail	Strong	Strong	Good

Figure 3.6: comparison between zxcvbn and popular websites' strength meters

regarded as a very strong password for all of the meters except zxcvbn. Even if the base word is uncommon, and it has some variations, it is not long enough to be considered so strong.

3. A password like **`correcthorsebatterystaple`** is not considered strong by most of the meters except zxcvbn, and it is not even allowed in some cases because it lacks numbers, Upper-cases or symbols.

The superiority of zxcvbn over the other meters in the table may seem like cherry picking, but the way zxcvbn is constructed explains these differences.

Matching

Enumerates all the (possibly overlapping) patterns it can detect. Currently zxcvbn matches against:

- **Dictionaries:** Common words the user is likely to use as password. Multiple dictionaries, in a simple .txt format can be used. In this work, we present a few: English words, Italian words, names and surnames, Burnett's 10,000 common passwords, words from tv and films. The match has an associated frequency rank, where words like the and good have low rank, and words like photojournalist and maelstrom have high rank. This lets zxcvbn scale the calculation to an appropriate dictionary size on the fly, because if a password contains only common words, a cracker can succeed with a smaller dictionary. For all dictionaries, match recognizes uppercasing and common `l33t` substitutions.
- **Spatial keyboard patterns:** Some users are likely to choose passwords based on spatial pattern. For instance a user could choose the first row of letters from right to left: `poiuytrewq` as they password. QWERTY keyboard, Dvorak keyboard, and keypad are considered.
- **Repeats**: Users are also prone to use repetition of characters, like `rrrr`.
- **sequences**: Numeric or alphabetic sequences like `123` or `fedcba`
- **years and dates**: The year or full date of a special event, like anniversary or birthday. Years from 1900 to 2019 are considered and dates in different formats. (3-13-1997, 13.3.1997, 1331997).

Entropy calculation of a single pattern

Depending on the type of matching, the entropy calculation is done differently, but for all the cases the idea is the same: How many different cases

a hacker would have to try before guessing the pattern? For example, for the repeat case, if the user chooses `zzzzz`, as it is repeated five times, and if we assume the hacker starts by the letter a, then the number of cases would be $N = 26 \times 5 = 130$. (The sequence the hacker would try is: `a, b, c, d...,z, aa, bb, cc,...,...,zzzz, aaaaa, bbbbb,....zzzzz`).

As the number of possible cases can be pretty large, the entropy is not given as a raw value but as $e = log_2(N)$, known as the entropy bits, and in some cases as $f = log_{10}(N)$, known as the log entropy. In the example, entropy bits: $e = log_2(130) = 7bits$.

The entropy bits and log entropy are related by:

$$N = 2^e = 10^f$$

$$f = e \times log_{10}(2)$$

$$e = f \times log_2(10)$$

Minimum entropy search of whole password

"Given the full set of possibly overlapping matches, the algorithm finds the simplest (lowest entropy) non-overlapping sequence. For example, if the password is damnation, that could be analysed as two words, dam and nation, or as one. It's important that it be analysed as one, because an attacker trying dictionary words will crack it as one word long before two.

zxcvbn calculates a password's entropy to be the sum of its constituent patterns. Any gaps between matched patterns are treated as brute-force "patterns" that also contribute to the total entropy. That a password's entropy is the sum of its parts is a big assumption. However, it's a conservative assumption. By disregarding the "configuration entropy" — the entropy from the number and arrangement of the pieces — zxcvbn is purposely underestimating, by giving a password's structure away for free: It assumes attackers already know the structure (for example, surname-bruteforce-keypad), and from there, it calculates how many guesses they'd need to iterate through."[18]

From entropy bits to rank and estimated crack time

To estimate the cracking time, it is necessary to make some assumptions about what kind of attack will be subjected the user. zxcvbn considers four possible scenarios according to the number of attempts/time the hacker can do:

1. **Online throttling (100 per hour):** Online attack on a service that ratelimits password authentication attempts.
2. **Online no throttling (10 per second):** Online attack on a service that does not ratelimit or where an attacker has outsmarted ratelimiting.
3. **Offline slow hashing (1e4 per second):** Offline attack. assumes multiple attackers, proper user-unique salting, and a slow hash function with moderate work factor, such as bcrypt, scrypt, PBKDF2.
4. **Offline fast hashing (1e10 per second):** Offline attack with user unique salting but a fast hash function like SHA1, SHA256 or MD5. A wide range of reasonable numbers anywhere from one billion to one trillion guesses per second, depending on number of cores and machines. Ballparking at 10B per sec.

zxcvbn then ranks a password with a security level from 0 to 4 according to its entropy value:

- **Level0 if ($N < 10^3$):** Too guessable, risky password
- **Level1 if ($N < 10^6$):** Very guessable, protection from throttled online attacks.
- **Level2 if ($N < 10^8$):** Somewhat guessable, protection from unthrottled online attacks.
- **Level3 if ($N < 10^{10}$):** Safely unguessable, moderate protection from offline slow-hash scenario.
- **Level4 if ($N > 10^{10}$):** Very unguessable: strong protection from offline slow-hash scenario

Where N is the number of possibilities a hacker would have to try for crack the password. So for instance, if the password is level 2, it could be cracked in around 10^8 guesses.

For Level0 the above rule in terms of the entropy bits is $e < log_2(10^3)$. In terms of the log entropy bits, it simply is $f < 3$.

The level and estimated crack time for each type of attack is presented to the user. With this information, the user will, hopefully, choose a Level4 password. Additionally, the user also receives feedback about how the password was cracked, so they know how to improve it.

3.3.6 PassPhrase Generator

From the previous two sections there seems to be a disagreement on what a good password looks like. PwGen can generate totally random passwords

or pseudo-random pronounceable passwords, but even the later go against what zxcvbn proposes: PassPhrases that are very easy to remember, but long enough to give excellent entropy results. To fill this gap, a PassPhrase generator that gives results along the lines of `CorrectHorseBatteryStaple` is used.

The PassPhrase generator developed by the author works by randomly picking out words from dictionary files. The user can tune the PassPhrase generation as follows:

- **Dictionaries:** The user must select appropriate dictionaries, containing a sufficiently large number of lines (larger than 10000) to ensure the picked words are really random. The English and Italian dictionaries used by zxcvbn are a good example. The user can work with as many dictionaries as desired, an the format must be one word per line. Only the first word of each line is counted, as everything after a space is trimmed.
- **Number of words:** The user can configure the number of words the generated PassPhrases are composed of. The recommended size is four, but it can be as long as the user wants.
- **Minimum Length of Words:** With this option is possible to select only random words whose length is higher than a certain value. This is to make sure the resulting PassPhrase is too short and therefore too insecure. The drawback here is that the higher the selected threshold, the fewer the available words in the dictionaries.
- **Only use infrequent words:** If the dictionaries follow the same format as those used for zxcvbn, that is, the words are ordered by frequency, having the most uncommon words in the lower part of the dictionary, the user can then ask to generate PassPhrases containing only unusual words. The drawback here is, again, fewer words to choose from. The percentage of words that are used is configurable.
- **Capitalize first letter:** To make the PassPhrases more readable, the first letter of each word can be capitalized.

3.3.7 Device side development: Eclipse

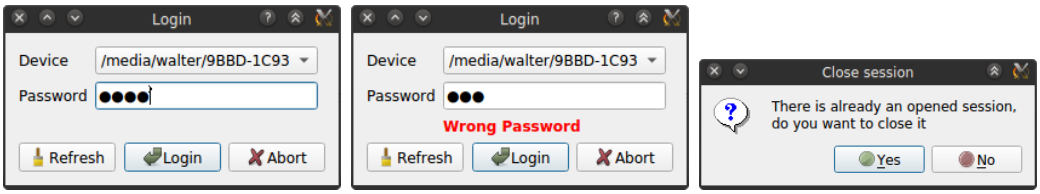
3.4 Implementation

In the following sections each of the elements and functionalities of the application will be explained, how they were implemented, some interesting pieces

of code and examples of use.

3.4.1 User authentication

When the user starts the application, the first window to appear is the Login Dialogue, shown in figure 3.7a. In it the user is asked to enter the login pin and by clicking accept the Challenge-Based Authentication process between the SEcubeWallet application and the SEcube™ chip starts. If the authentication fails because the entered pin is wrong, the message in 3.7b is shown. If it fails because there was already an opened session, the confirmation dialogue shown in 3.7c appears. If the authentication is completed successfully the user is granted access to the main window.



(a) Login Dialogue (b) Wrong Password (c) Already an open session

Figure 3.7: Login Dialogue and possible outcomes

The authentication process begins with the discovery of SEcube™ devices connected to the PC. This is achieved using Level0 APIs as seen in the listing 3.1. Each discovered device is added to the `QComboBox` displayed in the login dialogue and to a `QList`.

The user then selects one of the discovered devices using the `QComboBox`, enters their login pin and clicks accept. This triggers the listing 3.2. The first step is to open the device communication using the Level0 function `L0_open`. Then `L1_login` starts the actual challenged based authentication using the login pin entered by the user. Using this type of authentication ensures the login pin is never communicated between the devices and stolen with a physical attack on the USB cable. Rather, a random number is generated in the host and transmitted to the device. The login pin is then used in both the host and the device to encrypt this random number using a `pbkdf2` function. The resulting key on the device is sent to the host, who compares it with its own key. If they are the same it means the login pin entered by the user is equal to the login pin stored in the device, and the only information transmitted are random numbers that an attacker cannot understand. (The

Listing 3.1: Connected Devices discovery

```
//// *** variables declaration ***
se3_disco_it it;
QList<se3_disco_it> device_found;
QComboBox* chooseDevice;
bool found = true;

//*** Refresh button slot *****
L0_discover_init(&it); //initialize iterator
while((found = L0_discover_next(&it))){ //move to next device
  chooseDevice->addItem(QString::fromLocal8Bit(
    it.device_info.path, -1)); //add to GUI
  device_found.push_back(it); //add to QList
}
```

actual authentication procedure implemented is a little bit more complex, but is based on the same idea described here).

After both device and host have the same key, it is used to encrypt the communication channel. A token is generated in the device and transmitted on the encrypted channel to the host. This token is a random number, and is used from that point on to validate any communication between host and device. The SEcubeChip does not accept any command from the host if the token it sends is not the correct one. The only command accepted without a token is off course, login. In the logout procedure, this token is cleared (set to zeros), so a login later on is possible.

One problem found during the development of the application is the follow-in: If after login in, the SEcubeWallet application crashes, the logout command, that is usually issued when closing, is never executed, and the device remains with an active token value. Therefore, when the user launches the application again, the login will fail, because the device expects a token value from the host. To solve this issue a few options were considered:

- Make sure the application never crashes. Because software applications are rarely completely bug-free, and even if they are, an external problem like a bug in the OS can make them crash, this option is not feasible.
- Make sure the logout command is issued even in the case the application crashes. This option sounded promising, and a two-process idea was even developed. Process 1 is only in charge of calling Process 2, in which the actual application was executed. if Process 2 crashes, Process 1 remains

Listing 3.2: Open device and try to login

```
// *** variables declaration ***
//use selected index at QComboBox to retrieve 'it' from QList.
int device_index = chooseDevice->currentIndex();
se3_disco_it it = device_found.at(device_index);

se3_session s;
se3_device dev;
int ret;
bool logout = false; //if true, L1_login logs out first

//*** Accept button slot *****
if(!dev.opened) // open communication with device
  if((L0_open(&dev, &(it.device_info), SE3_TIMEOUT))!= SE3_OK)
    exit(1); //error

ret = L1_login(&s, &dev, pin, SE3_ACCESS_USER, logout); //login

if (ret != SE3_OK){     //error at login
  if (ret == SE3_ERR_PIN) //The password is wrong
    show_wrong_pass_messsage;

  else if (ret == SE3_ERR_OPENED){
    // there is already an opened session, ask user if he wants
       to close it
    if(confirmation_dialog_reply == Yes){
      logout = true;
      call_this_function_again;
      //next time L1_login will close the existing session
    }
  }else
    exit(1);//other error
}else
  accept();//All ok, go to main window
```

alive and performs the logout procedure. To make this work, shared memory was used to communicate the session variable (were the token is stored), so both processes could send commands to the device. The idea was latter on dropped because of two main reasons: First of all, it did not solve the case were the problem is external (OS bug), secondly, it was too complicated because of the sharing memory mechanism (The

session variable is a fairly complex structure, with a lot of pointers), and because the token was being shared by two processes, this could open another possibility of attacks.

- As the two previous ideas failed, it was decided that a small modification to the login behaviour on the SEcube™ firmware was necessary. The modification consist on letting the login function clear the token field if necessary. This does not compromise the security of the system because access to the chip is only granted if the login pin entered by the user is the right one. One concern that may rise is that, while an application is using the SEcube™, another one could close the session by issuing a Login command, but this is not possible because the `L0_open` function only allows one process to communicate with the chip at a time, using a file locker in the .se3magic file saved in the SEcube™ SDcard.

The new behaviour is implemented in listings 3.3. If after a crash the session in the SEcube™ remains open, and the host tries to login again, the SEcube™ returns the new error code `SE3_ERR_OPENED`. The host then can decide if it wants to force the SEcube™ to close the opened session so it can login, with the new command `SE3_CMD1_LOGOUT_FORCED`, which forces a logout without checking the token. After this command the host can login as usual. This steps are included in the host side `L1_login` function, which now has an additional parameter to control whether to force a logout or not. This parameter is the one used in listings 3.2, set to true when the user clicks YES in the confirmation dialogue asking whether or not to close the previous session.

3.4.2 Main Window

The SEcubeWallet GUI's main window developed using Qt is shown in figure 3.8.

The main window is composed of the following elements:

- **Table View:** Used for displaying the wallet entries. It resizes smoothly with the window, can be ordered by any of the columns, and the passwords are hidden by default but can be shown if the user wants to.
- **Filters:** The user can search in each of the table's columns using filters. These filters are implemented inside a separate container, but they resize together with the table.

Listing 3.3: Modification in SEcubeFirmware, file se3_cmd1.c

```
if (se3c1.login.y) { // if there is already an opened session
  if (memcmp(se3c1.login.token, req_params.token,
      SE3_L1_TOKEN_SIZE)) { //and token mismatch
    if (req_params.cmd==SE3_CMD1_CHALLENGE)//someone (maybe same
        user after a crash) trying to login.
      return SE3_ERR_OPENED;//notify host there is already an
         opened session, if host wants to continue, it will call
          SE3_CMD1_LOGOUT_FORCED
    else if (req_params.cmd==SE3_CMD1_LOGOUT_FORCED)//if the
       user agreed to close the existing session by forcing a
       logout
      req_params.cmd=SE3_CMD1_LOGOUT;//call logout as usual
    else
      return SE3_ERR_ACCESS;
  }
}
```

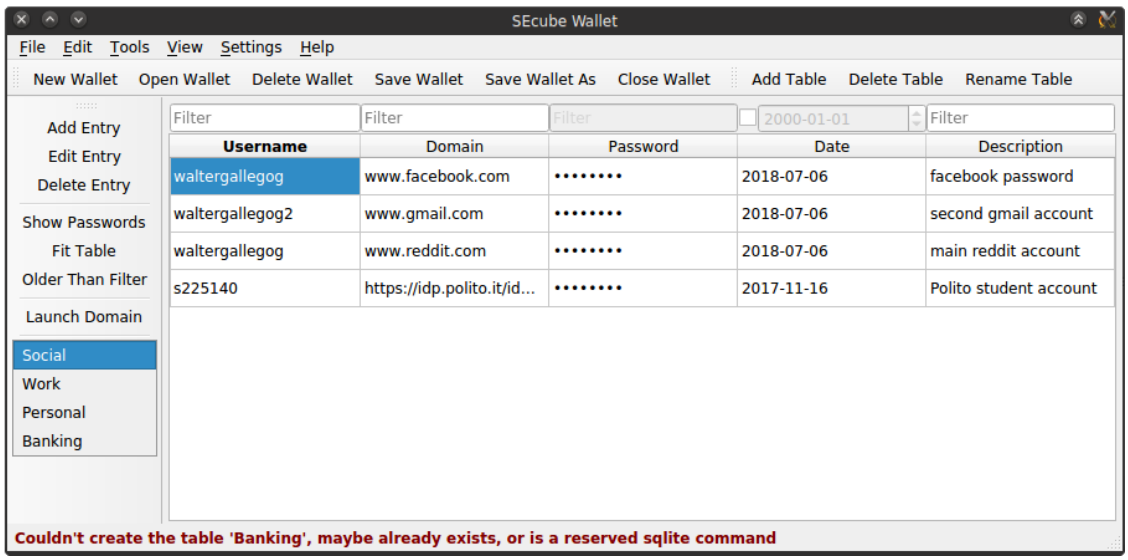


Figure 3.8: SEcubeWallet main window

- **Entries Tool Bar:** It is positioned to the left of the table. It has the actions: add/edit/delete entries, show passwords, fit table, change date filter, launch domain and select table.
- **Tables Tool Bar:** It is positioned to the top right of the table. It has the actions: add/rename/delete table.

- **Wallets Tool Bar:** It is positioned to the top left of the table. It has the actions: new/open/delete/save/save as/close Wallet. All of the above Tool Bars are movable.
- **Menu Bar:** It is positioned at the top of the window. It contains all the previous actions, plus preferences and help.
- **Status Bar:** Positioned at the bottom of the window, it is used to display some success/error messages to the user and the current wallet name.

3.4.3 Wallet actions

The actions regarding wallets: New, Save, Save As, Open, Close and Delete are explained in detail in the following.

New Wallet action

When the user triggers the the `New_Wallet` action, the first step to execute is to check if there is another wallet opened and if it has unsaved changes. If, so the confirmation dialogue in figure 3.9 is shown, so the user can decide whether to save the changes, discard them, or cancel the creation of a new wallet.

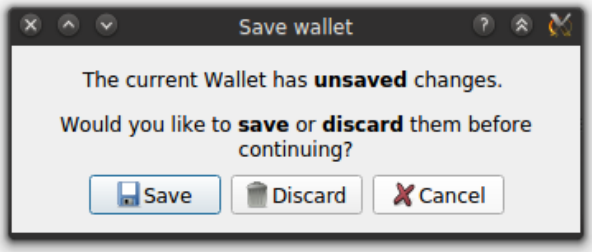


Figure 3.9: Save Confirmation dialogue

In case the user clicks `Save`, the `Save_Wallet` action is triggered before continuing. `Discard` continues without saving, and `Cancel` returns without doing anything.

If the process continues, the next step is to close any previous in-memory database handlers, save the `table_view` geometry (if any), and open a new in-memory database using the Qt class `QSqlDatabase`, as seen in listings 3.4.

Listing 3.4: New in memory database

```
QSqlDatabase dbMem; //The database handler, declared in header

//Check if SQLite is installed on OS
if(!(QSqlDatabase::isDriverAvailable("QSQLITE")))
  exit (1); //the application does not work without SQLite

if (dbMem.isOpen()){
  save_table_geometry;
  dbMem.close();//close any prev. opened database
}

dbMem = QSqlDatabase::addDatabase("QSQLITE");
dbMem.setDatabaseName(":memory:"); // in-memory database
if(!dbMem.open()){
  return;//Error opening, do nothing
}
```

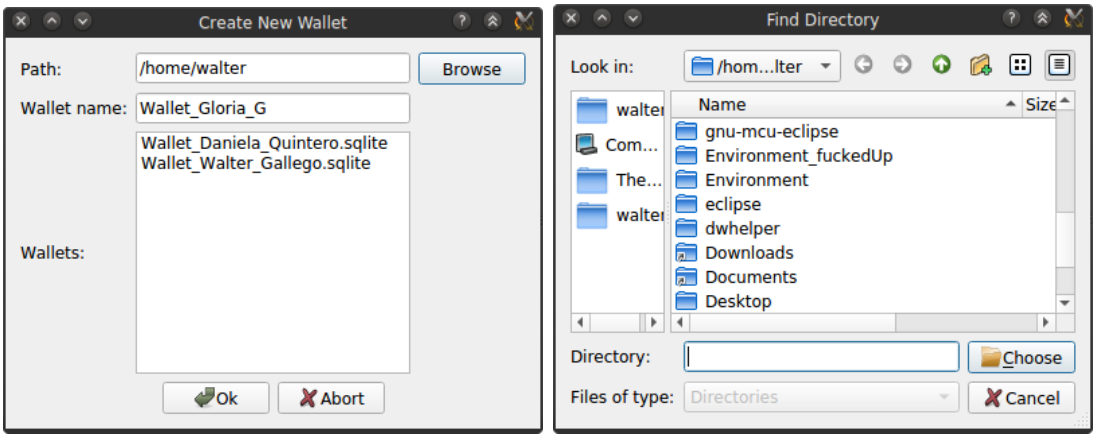
As explained before the in-memory data base is used for editing. It has the advantage of being fast because there is no access to the hard disk, and secure, because all the data is in the application memory space, and therefore is protected by the OS.

The last step is to update the GUI state, by enabling some action like `Add_Table` and `Save_Wallet`, and disabling others, like `Delete_Table` and `Rename_Table`.

Save Wallet action

To write the wallet contents to the disk, it is necessary to have a filename, so the first step is to check if the user already entered one (from previous saves). If not, with the dialogues in figure 3.10 the user can choose the directory and the filename to save.

The need for two dialogues instead of a regular file browser comes from the fact that the chosen filename will not be readable from the OS, since `SEfile` also encrypts it. Similarly, wallets already saved in the directory cannot be displayed with a regular file browser, so it is necessary to use the `SEfile` function `secure_ls` and display its output in the list seen in figure 3.10a. The declaration of this function is in listings 3.5. To chose the working directory, it is enough to use the `QFileDialog` class. If the user wishes they



(a) User can enter a New name. Current wallets are displayed

(b) if `Browse` is clicked a `QFileDialog` is launched

Figure 3.10: Save Wallet dialogues

chose an existing filename and can overwrite the correspondent wallet.

Listing 3.5: `secure_ls` declaration

```
1  /* This function identifies which encrypted files and encrypted
       directories are present in the directory pointed by path
      and writes them in list. It only recognizes the ones
      encrypted with the current environmental parameters.*/
2
3  uint16_t secure_ls(     //returns 0 in case of success
4    char *path,           //[in]Path to the directory to browse
5    char *list,           //[out]Allocated array to store filenames
6    uint32_t *list_length//[out]Num of char written in list
7  );
```

After having a filename the next step is to read all of the tables in the current in-memory database, row by row. For each row a `SQLite` statement of the form `INSERT INTO table VALUES(user, dom, pass, date, desc)` is created. All of them are merge into a single statement which is executed into the secured in-disk database. This ensures only one access to the SEcube™ and disk. This process is somewhat slow and the GUI is disabled while it is performed. A simplified version of the code is shown in listing 3.6

Listing 3.6: simplified Save process

```
sqlite3 *dbSec;     //Secure database declaration, in header
QSqlDatabase dbMem;//The database handler, declared in header
QSqlQuery query;    //To exec SQLite statements, dec. in header

//Create SQLite DB, with filename specified by user If SEcube is
    connected, the resulting file is encry.
sqlite3_open_v2 (fileName.toUtf8(),
                 &dbSec,
                 SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE ,
                 NULL)

QString finalSql; //To Merge all the SQLite statements.
static const QString insert =
  QStringLiteral("INSERT INTO '%1' VALUES(%2);"); //statement

dbMem.tables(QSql::Tables); //list of the tables in in-memory DB
tables.prepend("NoEmpty");  //Workaround: prepend Empty table
foreach (const QString table, tables) { // loop all the tables
   QString sql= "create table '"+table+//Create table statement
                "'(id integer primary key, "
                "Username TEXT, "
                "Domain TEXT, "
                "Password TEXT, "
                "Date TEXT, "
                "Description TEXT );" ;
  sqlite3_exec(dbSec, sql.toUtf8(), NULL, 0, &zErrMsg);// exec

  if (table=="NoEmpty"){//just an empty table
    set_values_to_empty;
    finalSql += insert.arg(table).arg(values.join(", "));
  }else{
    query.prepare(QString("SELECT * FROM [%1]").arg(table));
    query.exec()
    while (query.next()){ //row by row
      values = query_read_row();
      finalSql += insert.arg(table).arg(values.join(", "));
    }
}
//single write into secure database, fill the tables
sqlite3_exec(dbSec, finalSql.toUtf8(), NULL, 0, &zErrMsg);
```

One problem found during the open process of a secured in-disk wallet is that the first table is always corrupt and gives the error: `database disk image is malformed`. The error only occurs when using the SEcube™ version of the SQLite library. Because it was impossible to find the origin of the error, it was decided to use a workaround: In the save wallet process, an empty table is inserted at the beginning of the in disk database (see line 16 in listings 3.6). When opening the wallets, the empty table is simply ignored. With this, the real tables are always correctly read an the application works as intended.

Save Wallet As action

This action is very simple, it just clears the current filename (if any), and calls the `Save_Wallet` action; as there is no filename, the user is forced to enter a new one. The only point to be careful about is that, in case the `Save_Wallet_As` process is aborted, the previous filename needs to be recovered, so before clearing, the filename is temporary stored in case it is needed.

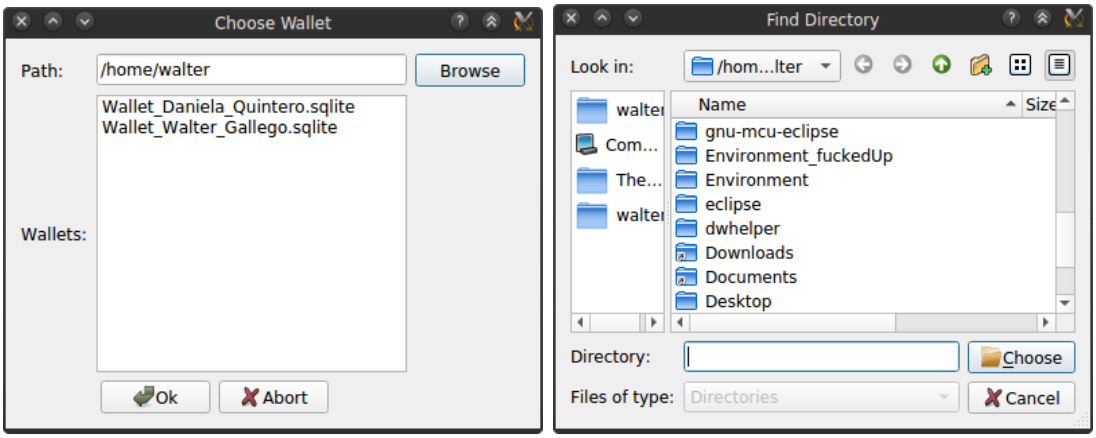
Open Wallet action

Similarly to the `New_Wallet` action, the first step is to check for unsaved changes and ask the user if save them, discard them or cancel, with the dialogue in figure 3.9

If the user decides to continue, the dialogues in figure 3.11 allow them to choose from the list the wallet to open.

The application proceeds doing the inverse process to the `Save_action`, that is, read all the tables from the secure in-disk database and create an in-memory database with this data. To do so, the listings in 3.7 is used. First, the in-disk data base is opened as read only, and a list named `tables` with the existent tables in it is generated. Then an in-memory database is created. Finally, for each table in `tables`, its contents are read, a correspondent table is created in the in-memory database, and the later is populated with the read contents from the in-disk DB.

The `SELECT FROM` statements in `sqlite3` require the use of a *callback* function, which is called for every result row, and receive the actual data from the data base as an `argv[]` argument. In the open process two of these functions are needed, shown in listing 3.8. In `create_TableList`, the list `tables` is build by simply reading the only element in the `argv[]` array, as



(a) User chooses the wallet to open from the list

(b) if `Browse` is clicked a `QFileDialog` is launched

Figure 3.11: Open Wallet dialogues

each row only consists of a table's name. In `populatetable`, the table in the in-memory DB is populated, row by row with each call. In this case the `argv[]` array holds the values in a single row coming from the in-disk DB.

Finally the GUI is updated, by enabling some elements and disabling others.

Close Wallet action

This action is very simple. As usual, before closing the current wallet a check for unsaved changes is performed, and the user is asked what to do with them using a confirmation dialogue. If the user decides to continue, the in-memory database handler is closed, and the table geometries are saved. Finally the GUI is updated.

Delete Wallet action

Deleting a wallet involves deleting an in-memory database and/or an in-disk database.

If only the in-memory database exists (user has not save it to disk yet), the wallet is simply closed, as in the previous section.

If there is no opened wallet, and the user wishes to delete an in-disk database, a select file dialogue equal to the one in the `Open_Wallet` action is shown, where the user can chose the wallet to delete. Then the wallet is

Listing 3.7: Simplified Open Wallet action

```
sqlite3_open_v2(fileName.toUtf8(),
                &dbSec,
                SQLITE_OPEN_READONLY, NULL)); //open in-disk DB

QString tableNames="SELECT name FROM sqlite_master "
                   "WHERE type='table' "
                   "ORDER BY name;";
sqlite3_exec(
  dbSec, tableNames.toUtf8(),
  callback_createTableList,//builds the 'tables' list
  this, &zErrMsg);

if (dbMem.isOpen()){
  save_table_geometry;
  dbMem.close();//close any prev. opened database
}
dbMem = QSqlDatabase::addDatabase("QSQLITE");
dbMem.setDatabaseName(":memory:");
dbMem.open();
query = QSqlQuery(dbMem);

foreach (const QString table, tables){
  QString sql = "create table '"+table+
                "'(id integer primary key, "
                "Username TEXT, "
                "Domain TEXT, "
                "Password TEXT, "
                "Date TEXT, "
                "Description TEXT );";
  if (table!="NoEmpty"){ //Workaround: ignore empty
    query.prepare(sql);
    query.exec(); //create table in in-mem DB
  }
  QString SqlStatement =
    QStringLiteral("SELECT * FROM '%1';").arg(table);
  sqlite3_exec(
    dbSec, SqlStatement.toUtf8(),
    callback_populateTable, //populates in-mem DB
    this, &zErrMsg);
}
```

Listing 3.8: Callback functions for Sqlite3 SELECT

```
//Build TableList from in-disk DB
callback_createTableList(int argc, char**argv, char**azColName){
  tables << argv[0]; //only one arg, the table name
  return 0;
}
//fill 'table' in in-mem DB with data from in-disk DB
callback_populateTable(int argc, char **argv, char **azColName){
  if (table=="NoEmpty") // we dont want NoEmpty in the in-mem db
    return 0;
  int i;

  static const QString insert =
    QStringLiteral("INSERT INTO '%1' VALUES (%2);");

  QStringList values;
  QString aux;
  for(i = 0; i<argc; i++){//argv holds values from a single row
      aux = argv[i];
      values << "'"+aux+"'";
  }
  query.prepare(insert.arg(table).arg(values.join(", ")) );
  query.exec();
  return 0;
}
```

deleted as shown in listing 3.9. First we obtain the encrypted version of the filename, using the SEcube™ API, and then the file in the disk is deleted using standard OS calls.

Listing 3.9: Delete an in-disk database

```
crypto_filename(fileName.toUtf8().data(),
                enc_filename, &enc_len
                );
QFile::remove(enc_filename); //OS call
```

If both in-disk and in-memory wallets are to be deleted, that is, if there is an opened wallet that has been already saved to disk, there is no need for a select file dialogue, as the filename is known from the `Save_Wallet` process.

The in-memory database handler is closed and the in-disk encrypted wallet file deleted as explained above.

In all the above cases, a confirmation dialogue ask the user if they are sure about deleting the wallet.

3.4.4 Table actions and display

Actions and classes regarding tables, they creation and display, are explained in the following sections:

Add Table action

With an in-memory wallet opened, the user can add a new table to it by simply entering a name. the listing 3.10 shows this process. With the `QIntputDialog` class, it is possible to ask the user for a name easily. If the name is valid, the sqlite query to add the table is executed. It may be the case the table already exists, in which case the query execution returns an error and the user is notified.

Listing 3.10: Add a New Table

```
1  bool ok;
2  QString tableName = QInputDialog::getText
3    (this, "New Table",
4    "Enter new table name",
5    QLineEdit::Normal,"", &ok);
6
7  if (!ok || tableName.isEmpty())
8    return;
9
10 QSqlQuery query(dbMem);
11 query.prepare("create table '"+tableName+
12                "'(id integer primary key, "
13                "Username TEXT, "
14                "Domain TEXT, "
15                "Password TEXT, "
16                "Date TEXT, "
17                "Description TEXT )");
18
19 if (!query.exec()){
20   return; // maybe a table with that name already exists, or is
       a reserved SQLite command
```

If the table is the first one to be added in the wallet, a `table view` and `filters` are created and added to the GUI, in order to display the table. Otherwise, the `table view` is just updated. These elements are explained in details in the upcoming sections.

Delete Table action

To delete a table it is enough to use the SQLite `DROP TABLE` command as seen in listings 3.11. With the `QMessageBox` the user is asked to confirm before deleting.

After deleting, the `table view` needs to be updated, to show the next table in the wallet. If there are no more tables, the view is simply hidden, and other GUI elements like `Add Entry` disabled.

Listing 3.11: Delete Table

```
1  reply = QMessageBox::question
2            (this,
3            "Delete",
4            "Are you sure you want to delete Table "+tableName,
5            QMessageBox::Yes|QMessageBox::No);
6  if (reply == QMessageBox::No)
7    return;
8
9  QSqlQuery query(dbMem);
10 query.prepare("DROP TABLE '" + tableName+"'");
```

Rename Table action

To Rename a table the SQLite command `ALTER TABLE RENAME TO` is used as seen in listing 3.12. After this, the `QComboBox` used to select the current table being displayed is updated to reflect the name change.

Select Table

To allow the user to select one table to display out of the existent ones in the current Wallet, a `QComboBox` is added to the Entries Tool Bar. Each time a Wallet is opened/closed, or a table is added/renamed/deleted, the items in the QComboBox are updated accordingly. When the selected item

Listing 3.12: Rename Table

```
1 QSqlQuery query(dbMem);
2 query.prepare("ALTER TABLE '" + currentName + "' RENAME TO '"+
     newName+"'");
3 query.exec();
```

in the QComboBox changes, the tableView update procedure explained in the following section is triggered.

Table display

To display the entries in each of the Wallet tables, a **Model/View** architecture was followed. From the official Qt documentation [8]:

"Model-View-Controller (MVC) is a design pattern originating from Smalltalk that is often used when building user interfaces. In Design Patterns, Gamma et al. write:

MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse.

If the view and the controller objects are combined, the result is the **model/view** architecture. This still separates the way that data is stored from the way that it is presented to the user, but provides a simpler framework based on the same principles."

"The model communicates with a source of data, providing an interface for the other components in the architecture. The nature of the communication depends on the type of data source, and the way the model is implemented.

The view obtains model indexes from the model; these are references to items of data. By supplying model indexes to the model, the view can retrieve items of data from the data source.

In standard views, a delegate renders the items of data. When an item is edited, the delegate communicates with the model directly using model indexes."

Figure 3.12 (taken from the Qt documentation), shows the way these three elements interact between them and with the data.

In the SEcubeWallet application these elements correspond to:

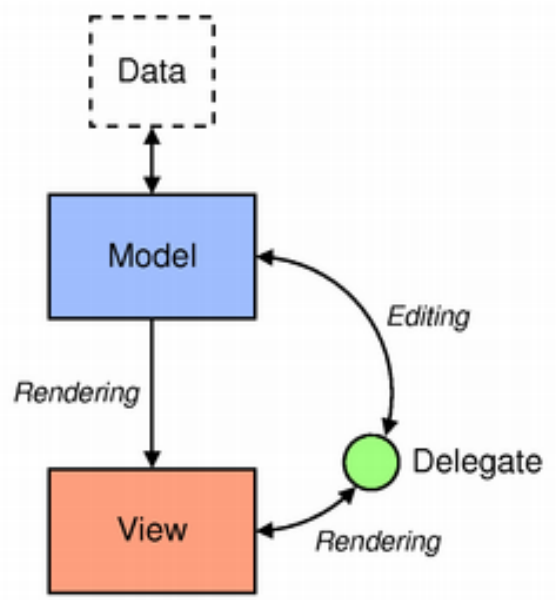


Figure 3.12: The Qt Model View Architecture

- **Data:** The data coming from the database. As we are displaying only one table at a time, the data actually corresponds to a single table.
- **Model**: `QSqlTableModel`, a Qt class which provides an editable data model for a single database table.
- **Proxy Model**: `MySortFilterProxyModel`, a class written by the author inheriting from the Qt `QSortFilterProxyModel` class. It implements custom filtering for each of the columns, in particular for the date.
- **View**: `MyQTableView`, a class written by the author inheriting from the Qt `QTableView` class. `MyQTableView` reimplements the `resizeEvent` to make sure the user has the best GUI experience. This new class allows for manual resizing of each column as well as automatic resizing when the widow size changes. Also, there is a button that allows the user to fit the table in the available space by giving all the columns the same width.
- **Delegate**: `PasswordItemDelegate`, a class written by the author inheriting from the Qt `QStyledItemDelegate` class. It is used to implement the show/hide password functionality.

There is one element not mentioned in the model/View architecture: the **Proxy Model**. It will be explained in details in the filters section. The

Listing 3.13: Model/View architecture implementation

```
// Create and configure model to access data in table
model = new QSqlTableModel;
model->setTable(tableName);
model->select();//update the model selection
model->setEditStrategy(QSqlTableModel::OnManualSubmit);
  //Changes will be updated manually by calling submitAll()

//Create ProxyModel for filtering
proxyModel = new MySortFilterProxyModel(this);
proxyModel->setSourceModel(model); //connect proxyModel to Model

//Create and connect delegate to hide passwords
passDelegate=new PasswordItemDelegate(this);
ui->tableView->setItemDelegateForColumn(PASS_COL, passDelegate);

//Connect to table view, resulting in:
//Sql <--> Model <--> ProxyModel <--> Delegate <--> TableView
ui->tableView->setModel(proxyModel);

//Configure table
ui->tableView->//Hide SQLite ID column, not useful to user
  setColumnHidden(IDENT_COL, true);
ui->tableView->//Hide row header, not useful
  verticalHeader()->hide();
ui->tableView->//To make the table view not editable
  setEditTriggers(QAbstractItemView::NoEditTriggers);
ui->tableView->//To allow only one row  selection.
  setSelectionBehavior(QAbstractItemView::SelectItems);
ui->tableView->//So we can edit one entry per time
  setSelectionMode(QAbstractItemView::SingleSelection);

ui->tableView->show();// show table
```

delegate will be explained in the Show Password section.

In listing 3.13 the model view architecture is implemented. Model, Proxy Model and Delegate are interconnected among them, with the database table and with the GUI's table view. Additionally some display tweaks are made, for instance the ID column, vertical header and passwords are hidden.

When it is necessary to change the table to be displayed, either because the user changes walllet or table, or deletes one, the listing 3.14 is used. Only

the **model** needs to be updated, because it is the one connected to the data. The table view geometry needs to be saved and restored, because with the data change, the columns are automatically resized by Qt to fit the new data, which is visually annoying for the user.

Listing 3.14: update the model/view

```
1 save_table_geometry;//each column width
2 model->setTable(tableName);
3 model->select();
4
5 restore_table_geometry;
6 ui->tableView->setColumnHidden(IDENT_COL, true);//required
```

Show Passwords action

To show the passwords it is enough to NOT use a delegate for the correspondent column. Conversely, to hide them again, the delegate is used, as seen in listing 3.15. The delegate `displayText` method definition is also shown. Its job is to return the bullet character eight times instead of the actual item value.

Filters

The filters implementation comprise two classes. The first one is related to the GUI, and its job is to align each of the filters with its corresponding table column. The second class is related to the data filtering per se.

The **filters alignment** is based on the `ColumnAlignedLayout` class by sashoalm [5]. This class is a Layout that inherits from the `QHBoxLayout` Qt class. It reimplements the `setGeometry` method to reposition each of the elements in the layout to follow the correspondent column in the tableView that is being tracked. This implementation allows to add any type of widget to the layout, as long as the number of widgets equals the number of tracked columns. The class definition is shown in listing 3.16.

To connect the aligned layout to the table view, the listing 3.17 is used. In it, an aligned filter object is created and is set as the Layout of a widget positioned over the tableView in the GUI. Then the aligned filter is set to track the tableView's `horizontalHeader`. Finally any geometry change in

Listing 3.15: Show/Hide Passwords

```
// *** in pasworditemdelegate.cpp ***
PasswordItemDelegate::PasswordItemDelegte(QObject* parent):
    QStyledItemDelegate(parent){}
//inherits from QStyledItemDelegate

QString PasswordItemDelegate::displayText
   (const QVariant &value, const QLocale &locale) const {
     return (QString("%1").arg(QChar(0x2022)).repeated(8));
} // just returns the bullet character 8 times

// *** in mainwindow.cpp ***
void MainWindow::on_action_Show_Passwords_toggled(bool show){

    passDelegate=new PasswordItemDelegate(this);
    if (!show)//do not show passwords: use passDelegate
      ui->tableView->setItemDelegatForColumn(
        PASS_COL, passDelegate);

    else{ //show passwords: do not use delegate
      QMessageBox::StandardButton reply;
      reply = QMessageBox::question(
                this,
                "Passwords",
                "Are you sure you want to show your passwords",
                QMessageBox::Yes|QMessageBox::No);
      if (reply == QMessageBox::No)
        return;//if error or cancel, do nothing

      ui->tableView->setItemDelegateForColumn(PASS_COL, 0);
    }
}
```

the `horizontalHeader`, or in the `horizontalScrollBar` calls the SLOT `invalidateAlignedLayout`, which in turns calls the `invalidate` method. This method resets the Layout's cached information, which forces a call to `setGeometry`. This ensures any resizing of the table's columns is reflected in the filters position.

The **data filtering** is done using a custom `SortFilterProxyModel`. For all the columns except the date, a simple case insensitive match is enough. For the date, using a button in the GUI, the user can choose among two

Listing 3.16: Aligned Filters definition

```
// *** filtersaligned.cpp ***
FiltersAligned::FiltersAligned(QWidget *parent) //constructor
: QHBoxLayout(parent){//inherits from horizontal Box Layout
  add_filters_to_layout; //add filters (most of them QLineEdits)
}
void FiltersAligned::setTableColumnsToTrack(QHeaderView *view) {
    headerView = view; //set tracked tableView
}
void FiltersAligned::setGeometry(const QRect &r){
  QHBoxLayout::setGeometry(r);

  int widgetX = parentWidget()->mapToGlobal(QPoint(0, 0)).x();
  int headerX = headerView->mapToGlobal(QPoint(0, 0)).x();
  int delta = headerX - widgetX;

  // repositioning
  for (int ii = 0; ii < headerView->count(); ++ii) {
    int pos = headerView->sectionViewportPosition(ii);
    int size = headerView->sectionSize(ii);
    auto item = itemAt(ii);
    auto r = item->geometry();
    r.setLeft(pos + delta);
    r.setWidth(size);
    item->setGeometry(r);
  }
}
```

options:

- **Exact Match filter:** In this configuration, the user enters the exact date they are looking for, using a `QDateEdit` input element.
- **Older Than filter:** Because this is a password wallet, looking for an exact date may not be very useful. Instead, knowing which passwords are older than a given amount of time, and have therefore expired is more helpful. For instance the user could be interested in updating their passwords each six months, so with this filter they can show only those passwords that need to be changed. In figure 3.13 an Older Than six months filter is shown (as of July the 13th 2018).

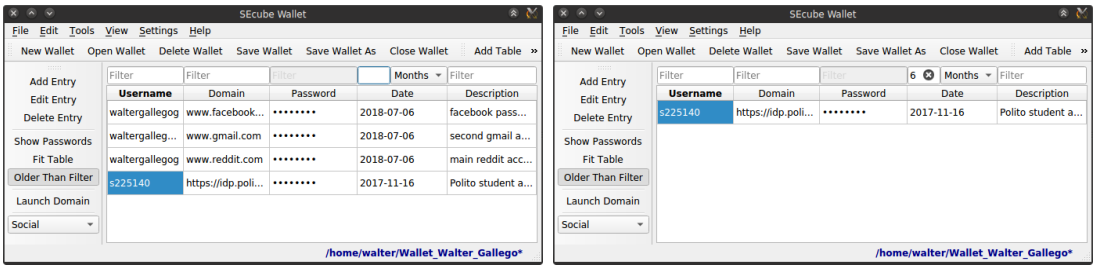
The filters implementation is very simple. It involves the reimplementation

Listing 3.17: Table View and aligned filters connection

```
// *** in mainwindow.cpp when the tableView is created:
  filters = new FiltersAligned();
  ui->filtersWidget->setLayout(filters);
  filters->setTableColumnsToTrack(
    ui->tableView->horizontalHeader());
  filters->setParent(ui->filtersWidget);

  connect(ui->tableView->horizontalHeader(),
          SIGNAL(sectionResized(int,int,int)),
          SLOT(invalidateAlignedLayout()));
  connect(ui->tableView->horizontalScrollBar(),
          SIGNAL(valueChanged(int)),
          SLOT(invalidateAlignedLayout()));

//the SLOT
void MainWindow::invalidateAlignedLayout(){
    filters->invalidate();
}//clears the cache, which forces a call to setGeometry
```



(a) No filter (b) Older Than 6 Months

Figure 3.13: Date Older Than filter

of the `filterAcceptsRow` method in the ProxyModel as in listings 3.18. This method is called each time the user changes one of the filters contents. The filters work concurrently, so the user can search in multiple columns at the same time. All of them except the date one accept `RegExp`, so a search like `google|gmail` is possible. The password filter is only enabled when the passwords are visible.

Listing 3.18: Filters implementation

```
bool MySortFilterProxyModel::filterAcceptsRow
(int sourceRow, const QModelIndex &sourceParent) const
{
  QModelIndex userIndex =
    sourceModel()->index(sourceRow, USER_COL, sourceParent);
  QModelIndex domainIndex =
    sourceModel()->index(sourceRow, DOM_COL, sourceParent);
  QModelIndex passIndex =
    sourceModel()->index(sourceRow, PASS_COL, sourceParent);
  QModelIndex descIndex =
    sourceModel()->index(sourceRow, DESC_COL, sourceParent);

  QModelIndex dateIndex =
    sourceModel()->index(sourceRow, DATE_COL, sourceParent);
  QDate thisDate = QDate::fromString(
    sourceModel()->data(dateIndex).toString(),format);

  return (//only return rows where the conditions are met
    sourceModel()->
      data(userIndex).toString().contains(userRegExp)          &&
    sourceModel()->
      data(domainIndex).toString().contains(domainRegExp)      &&
    sourceModel()->
      data(passIndex).toString().contains(passRegExp)          &&
    sourceModel()->
      data(descIndex).toString().contains(descRegExp)          &&
    (thisDate<=filterDate_older||!filterDate_older.isValid()) &&
    (thisDate==filterDate_exact||!filterDate_exact.isValid()) &&
  );
}
```

3.4.5 Entries actions

In this section, the implementation of the actions add/edit/delete entry are explained.

Add Entry action

The `AddEntry` class allows users to input a new entry to one of the tables, using the subwindow in figure 3.14

This subwindow is composed of:

(a) Passwords are shown and do not match

(b) Passwords are shown and do match

(c) Passwords are shown. zxcvbn library has not been compiled

Figure 3.14: Add Entry subwindow

- **Text input elements:** To enter the Username, Domain, Description and Password, five `QLineEdit` are used. The password needs to be entered twice to make sure is the desired one, and if it is not the same in both text fields, an error message is displayed. The date is not entered by the user, but generated automatically based on the system clock.
- **Show password checkBox:** By changing the echo mode in the password `QLineEdit`, it can be hidden or shown. It is hidden by default
- **Password Generator Button:** This triggers one of two available generators: PwGen or PassPhraseGen. These are explained in detail in sections 3.4.7 and 3.4.9
- **l33t buttons:** To increase the password strength the user can translate it to l33t. If they are not happy with the result, a button to reverse the action is also present. The l33t implementation is explained in section 3.4.6
- **Password Strength elements:** A Progress Bar used to display the password strength calculated with the zxcvbn library, a label to show some information about the strength, and a button to open a subwindow showing the details of these calculations. If the zxcvbn library has not been compiled by the user in the settings window, these elements are disabled. The zxcvbn library is explained in details in section 3.4.8
- **Bottom buttons:** The `ok` button is only functional when all the text fields (except for `description`, which is optional) are filled, and the two

passwords coincide. The `settings` button opens the settings subwindow so the user can customize the password generators or the strength estimator without having to close the `AddEntry` subwindow.

When the user clicks the `ok` button, the new entry needs to be added to the database. Instead of using SQLite commands, it is easier to rely in the high-abstraction level methods offered by the `model`, as seen in listing 3.19.

Listing 3.19: Add entry to database using model

```
AddEntry *add = new AddEntry(this);
add->exec(); //exec AddEntry subwindow
if(add->result()==QDialog::Rejected)
  return; // Error or cancel, do nothing

QSqlRecord rec = model->record(); // Temp entry
rec.setGenerated("id", false); // is managed by SQLite

//Get the values entered by user in AddEntry subwindow
rec.setValue("Username"   , add->getUser());
rec.setValue("Password"   , add->getPassword());
rec.setValue("Domain"     , add->getDomain());
rec.setValue("Description", add->getDescription());

rec.setValue("Date",        QDate::currentDate());//sys clock

int newRecNo = model->rowCount(); //insert at the end of table
if (!model->insertRecord(newRecNo, rec))
  return;
model->submitAll();// if insert ok, submit changes.
```

Edit Entry action

The user can edit any of the entries by selecting one of the cells in the `tableView` and clicking the `Edit Entry` button, or by double-clicking any of the cells. In either case, the data from the selected row is retrieved and passed to new `AddEntry` object using its constructor. In this way, the user is presented with an `AddEntry` subwindow where the input fields are already filled with the current data. The user can then modify and save them by clicking OK. In this process the `ProxyModel` is used instead of the `model`, because the former allows to identify the items selected in the `tableView`.

Delete Entry action

Deleting an entry is very simple. The row index of the selected cell is used in the `ProxyModel` method `removeRow(row)`, and the change is submitted with `submitAll`. Before deleting, the user is asked to confirm the action.

3.4.6 Other functionalities

In this section, a set of miscellaneous functionalities are explained.

Launch Domain action

Using the Qt function `QDesktopServices::openUrl(QUrl(domain))`, it is straight forward to open the domain of the entry selected by the user. It will be opened in the default web browser configured in the OS. The only detail to play attention is the format. If the domain entered by the user does not start with `http://` or `https://`, then `http://` is prepended. Also, if the entered domain does not contain any dots apart from the one in `www.`, then the most common top-level domain, `.com` is added. This will not work in all the cases, but should help fixing some of the domains when the user forgets to type the TLD.

Status Bar

The status bar, at the bottom of the main window, is used to display three types of information:

- **Wallet Name:** The wallet name (with full path) is permanently displayed at the right side of the status bar in color blue. In case the wallet does not have a name because the user has not saved yet, `unnamed` is displayed. When the wallet has unsaved changes, an asterisk is added to the end resulting in: `/absolute_path/wallet_name*`
- **Success:** After any save/open/delete operation, a black message is shown for two seconds in the left side of the status bar, informing the user the process concluded without issues.
- **Error/Warning:** In the eventuality of an error or warning during the execution of a command, for instance, if it is not possible to load the SQLite driver, a red message in the left side of the status bar informs the user about the issue.

Preferences Subwindow

This subwindow is accessible from the `Menu Bar` and from the `AddEntry` subwindow. It allows to customize the password generators and the strength estimator. Their available configurations are covered in their respective sections.

Environment subwindow

From the environment subwindow the user can select which of the keys and algorithms present in the SEcube™ chip to use for the encryption/decryption process.

Keys are used to divide wallets into categories (work, banking, social) and allow their sharing. A given user can have for instance one key that is shared among co-workers, so all of them can access work-related passwords using their respective SEcube™ devices. Similarly, other key can be shared with their family to access banking accounts passwords, and finally have a personal key that is not shared with anyone and is used to encrypt social media passwords.

The current SEcube™ firmware version only includes one algorithm for the data encryption/decryption process, but in the future it may provide more than one. Different algorithms offer different security characteristics that some advance user may find useful. For example one user could be concerned about some specific type of attack that requires an special algorithm, or regard one algorithm superior over the others.

Figure 3.15 depicts the environment subwindow.

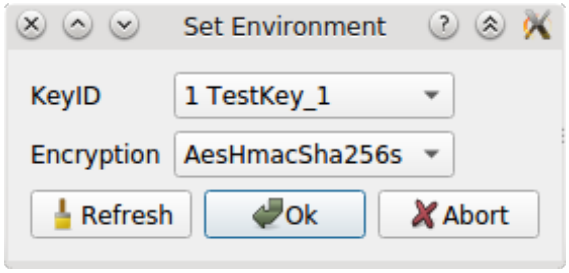


Figure 3.15: Environment subwindow

Help Subwindow

This subwindow teaches the user a few tips about how to use the application features, some of which may not be obvious, like the strength estimator or the date filter. It also gives some information about the application, like the author, year, and code sources.

l33t

This is a very simple converter that uses the capabilities of the `QString` class to replace each occurrence of the letters `o i z E A s b T B P` with the numbers `0 1 2 3 4 5 6 7 8 9` respectively, and the letter `l` with the character `|`. To un-l33t, the reverse process is made. It is worth noticing that password crackers (and strength estimators like zxcvbn) usually try an l33t dictionary too, so this option may increase the password strength, but not by too much. In table 3.1 a few passwords and their l33t version are compared. The strength measured with zxcvbn is reported.

Table 3.1: A few l33t examples

Password	**Log Entropy (Level)**
LeytonVariational	9.44 (Level 3)
Leyt0nVar1at10na|	10.22 (Level 4)
PenicuikCiting	9.09 (Level 3)
9en1cu1kC1t1ng	13.87 (Level 4)
LauraDrogheda	6.44 (Level 3)
LauraDr0gheda	6.74 (Level 4)

3.4.7 PwGen: Pronounceable Passwords Generator

As seen from previous sections, the PwGen program is open source and available in the official Linux repositories. A very simple way of including its functionalities into the SEcubeWallet application would be to use a `Qprocess` to call PwGen as an external program. Although is tempting to use this solution because of its simplicity, there are three drawbacks with this approach:

1. It would require for the user to install the PwGen program, as it is usually not included in common Linux distributions.
2. It would not be very portable, because even if there is a PwGen version for windows, the available version or input parameters could differ in different platforms.
3. Security could be compromised. As PwGen needs to communicate the generated password back to the SEcubeWallet application, an attacker could steal the password in this process.

For these reasons, it was decided to embed the PwGen sources directly into the application, this ensure the password never leaves the application memory space. To include the sources into the application, some slight modifications (mostly simplifications) to the PWGen `main()` function were necessary. This is because the original sources are intended for the use of PwGen as a independent console program called by users, so the `main()` contains code dedicated to parse the input arguments in the standard `argc argv[]` fashion. Only the `pwgen.c` and `pwgen.h` — files where the `main()` is implemented — were modified.

Options GUI

As all the options for PwGen besides the password length are yes or no questions, the checkable list shown in figure 3.16 is perfect for this purpose.

When the `OK` button is clicked, all the values are saved using the `QSettings` class, so they are available even after restarting the application.

Usage

When the `Generate Password` button in the `AddEntry` class is clicked, the settings values stored in with the `QSettings` class are read and from them a `char[]` with the PwGen options syntax is built. Before calling the Generator, it is necessary to allocate a memory space equal to the desired password length (in `char`) and pass this buffer as a pointer (`char*`) to PwGen. PwGen will write the generated password in this space. This steps can be seen in the listing 3.20

Results

A few examples of the resulting Passwords are show in table 3.2, with their respective zxcvbn Log entropy score and Level. As expected, the passwords

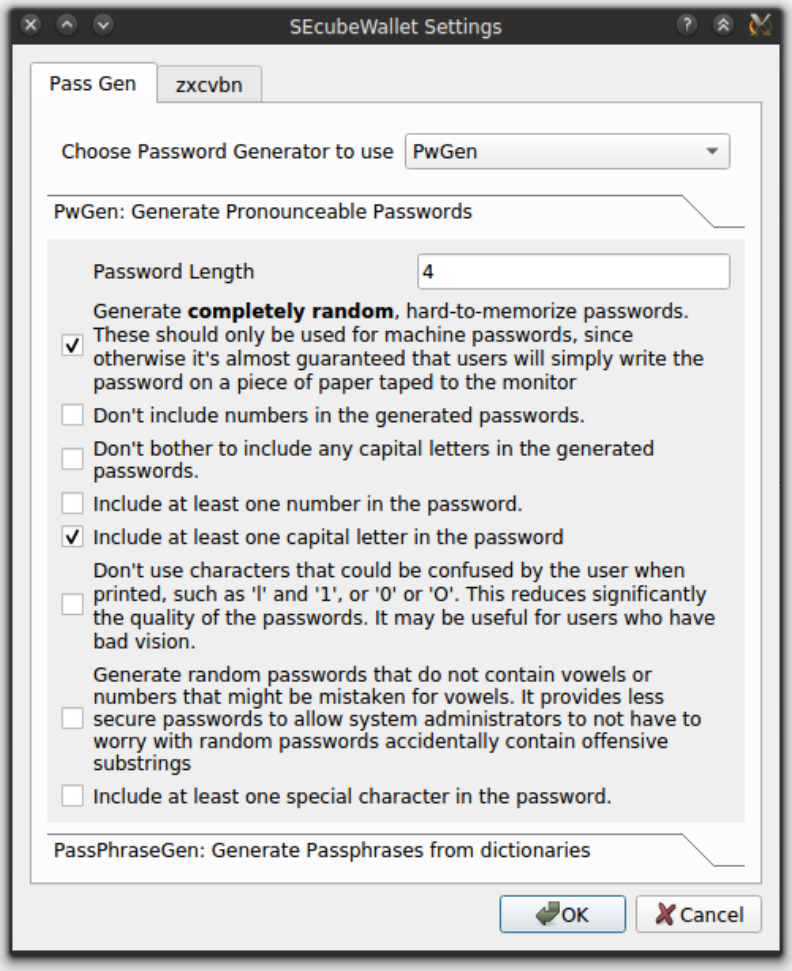


Figure 3.16: PwGen settings in the preference window

with highest entropy are those obtained using the `-s` (Random) and `-sy` (Random and especial characters) options. Even with only 6 characters it is possible to get a Level 4 password like `TBw4)9`. The drawback is, they are hard to remember and type.

On the contrary, a password obtained with the `-BA0` (No ambiguous, do not capitalize, no numbers, pronounceable) options, like `nofosootei`, only reaches a Level2, but it is very easy to remember and type.

Listing 3.20: PwGen call inside AddEntry

```
//read user settings (if existent)
if (settings.value("passGens/pwgen/1cap").toBool())
    options.append("c");
if (settings.value("passGens/pwgen/1num").toBool())
    options.append("n");
if (settings.value("passGens/pwgen/1spec").toBool())
    options.append("y");
if (settings.value("passGens/pwgen/noAmb").toBool())
    options.append("B");
if (settings.value("passGens/pwgen/noCap").toBool())
    options.append("A");
if (settings.value("passGens/pwgen/noNum").toBool())
    options.append("0");
if (settings.value("passGens/pwgen/noVow").toBool())
    options.append("v");
if (settings.value("passGens/pwgen/random").toBool())
    options.append("s");

//check if user entered an integer, if not, default is 16
if(settings.value("passGens/pwgen/len").toInt())
    length = settings.value("passGens/pwgen/len").toInt();

//allocate space for password
buf = (char*)malloc(length+1);
if(!buf){
  return;//error, could not allocate
}
//actual call to password generator
main_pwgen(
  options.length(),               //int, number of options
  options.toLatin1().constData(), //char *, options
  length,                         //password length
  buf                             //char *, to return the
     password
);
genPass = QString::fromLatin1(buf,length);
free(buf);
```

3.4.8 zxcvbn Password strength estimator

The original zxcvbn project, developed in CoffeeScript became so popular it was ported to a large variety of languages. In this work the C/C++ version

Table 3.2: A few PwGen generated passwords

Password	Length	Options	Log Entropy & Level
iesohGhai3	10	-	9.75 (Level 3)
ees0cooLo2	10	-	10.47 (Level 4)
dX042wKqlW	10	s	17.86 (Level 4)
@!,Q*l5}+H	10	ys	18.15 (Level 4)
TBw4)9	6	ys	11.62 (Level 4)
B7t34Lck	8	v	11.87 (Level 4)
nofosootei	10	BA0	6.50 (Level 2)

available at [17] was used. The files used in this project are:

- **zxcvbn.c** Main source file
- **zxcvbn.h** Main header file
- **dict-generate.cpp** Used for generating the dictionary sources
- **Makefile** To compile the dictionary generator and main program.
- **words-*.txt** A few examples of dictionary files in plain text format.

Besides source files, zxcvbn also needs to compile the dictionary files, but first lets define what is a dictionary, why are they important and why they need to be compiled (For general dictionaries only. User dictionaries are small and can be added at runtime).

General dictionaries

Dictionaries are a crucial part of the algorithm, because they are used estimate the security level of a password according to how common the used words (if any) are. A password containing words present in any of the dictionaries will be easier to crack as hackers will probably try out those specific words or a combination of them.

General dictionaries contain a large number of words that are useful for all users. Examples of these type of dictionaries (included in this work) are:

- 100000 English words from wikipedia.

- 88800 Last names from the US census database
- 39000 English words from tv and film from the wikiproject [16]
- 47000 Most common passwords from Burnett [22]
- 15480 Italian words from the BADIP project [3]
- 4276 Female names from the US census database
- 1220 Male names from the US census database

As the dictionary files in plain text are pretty large, the algorithm does not read from them directly. Instead, a `DicNodes` array is generated, using the tool `dict-generate`, and this array is compiled into the source code. To add their own dictionaries, the users need to make sure they are saved as plain-text, (.txt UTF-8), and stored into the zxcvbn directory. The files must have one word per line, with the first word being the most common one. So for instance, in the English dictionary the first word is `the` and the last one is `surma`. This is important as it is used to calculate the entropy of the passwords. A password containing the word `surma` is far more secure that one containing the word `the`.

Static Library vs Shared Library

Because the dictionaries are transformed into a source file and then compiled together with the main program, it is not possible to add, remove or modify dictionary files after the sources are compiled. Therefore the zxcvbn library can not be embedded into the SEcubeWallet application as a static library (or using the C sources directly), but rather, a **shared library** approach was followed, which allows the dynamic unload/update/load of the library. This has some performance penalties over static libraries, but it is the only way to give the users the possibility of customize the dictionaries as they please.

Compilation process

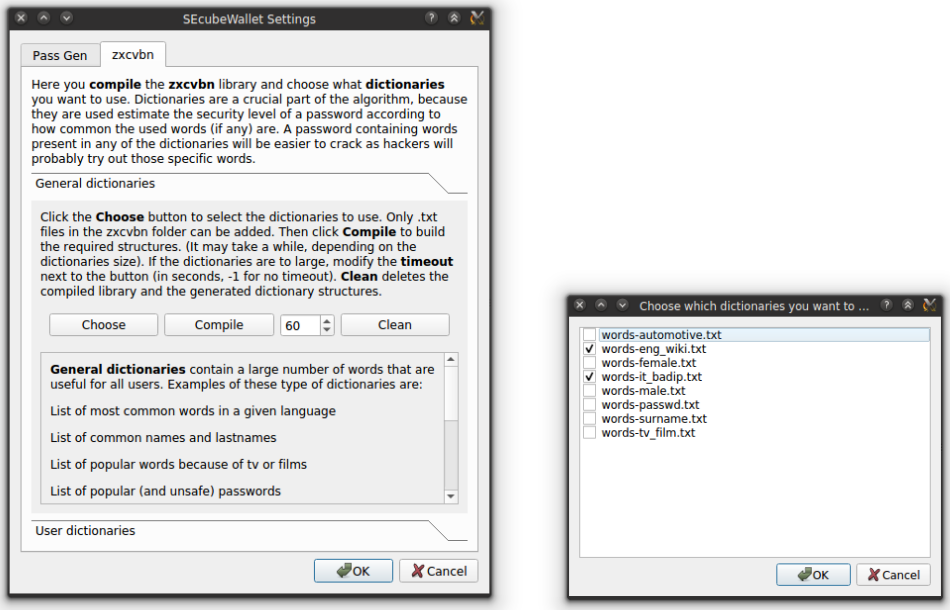
The steps performed by the `makefile` in order to compile the dictionaries and sources are:

1. Compile the source file `dict-generate.cpp` to obtain the dict generator executable`dictgen`.
2. Execute `dictgen` with the names of dictionaries to process as input argument. As a result the file `dict-src.h` is created.

3. Compile the files `zxcvbn.c` and `zxcvbn.h` together with the just generated `dict-src.h`, using the gcc flag `-fPIC` so the resulting object file `zxcvbn-inline-pic.o` is suitable for library inclusion.

4. Generate the shared library `libzxcvbn.so` from the object file. This is the library used by the SEcubeWallet sources.

The compilation process can be started by the user from the preference window, where they can also select the dictionaries to use, or clean the generated files. The compiling is made with OS calls, through the use of `QProcess`. To avoid the application from crashing or getting stuck, the `Qprocess` has a timeout. As the compile process may take a while depending on the dictionaries size, this timeout can be configured by the user. In figure 3.17 the GUI for these actions is shown.



(a) Tab dedicated to zxcvbn preferences

(b) Checkable dialogue to enable dictionaries, appears after clicking the Choose button

Figure 3.17: zxcvbn general dictionaries configuration

Dynamic Library loading

To manage the zxcvbn shared library at runtime it is possible to use the qt class `Qlibrary`, which provides access to the functionality in the library in a platform independent way. To use it, is necessary to pass as argument to the constructor the path to the library. Then load it and resolve the desired functions. If no errors are found, the functions can be used as usual. Finally unload the library when it is not needed any more. See Listing 3.21.

User Dictionaries

The user dictionary contain words that are relevant only to a specific user. For example, if the application is used to increase the strength level of passwords used by employees in a company, adding the company's name to the dictionary is a good idea. Furthermore, if the company works in the automotive business, related words as motor, aerodynamic, wheels etc. should be added. By adding those words to the user dictionary, the strength level of passwords using them will decrease, and so the user will be encouraged to never use words that are too easy to guess. The key to a good password is in its randomness. When a hacker is trying to crack one, they will for sure try words relevant to the target.

From the GUI the user can add words manually, or can load them from a text file, but as the words are saved as a simple array, the text file size should not be too large. For large files, it is better to add them as General dictionaries.

Estimator Usage

After the library is loaded and the functions resolved, to use the estimator one simply needs to call the main function `ZxcvbnMatch` whose declaration we see in Listing 3.22

To obtain the password strength level, it is necessary to compare the `zxcvbnMatch` return value (The entropy in bits) as seen in section 3.3.5. The strength level is shown to the user with a progress bar. To the user may be more relevant to see some estimates about how long it would take for an attacker to crack the password. This information can be obtained from the entropy, assuming some numbers for the attempts/time the attacker can perform. These results are shown in a table like the one in figure 3.18

Some interesting additional information can be obtained from `ZxcMatch_t ** Info`. By traversing the data in this pointer, it is possible to see how the

Listing 3.21: Qlibrary basic usage

```
1
2  /***** In header file *****/
3  //Main zxcvbn function type
4  typedef double (*ZxcvbnMatch_type)(const char *Passwd,
5                                     const char *UserDict[],
6                                     ZxcMatch_t **Info );
7  //Function used to free the Info structure
8  typedef void (*ZxcvbnFreeInfo_type)(ZxcMatch_t *Info);
9
10 QLibrary * zxcvbnLib = 0;
11 ZxcvbnMatch_type ZxcvbnMatch = 0;
12 ZxcvbnFreeInfo_type ZxcvbnFreeInfo = 0;
13
14 /***** In cpp file *****/
15 zxcvbnLib = new QLibrary(zxcvbn_lib_path);
16
17 if(zxcvbnLib->load()){
18   ZxcvbnMatch = (ZxcvbnMatch_type) zxcvbnLib->resolve("
       ZxcvbnMatch");
19   ZxcvbnFreeInfo =(ZxcvbnFreeInfo_type) zxcvbnLib->resolve("
       ZxcvbnFreeInfo");
20 }
21
22 if (!ZxcvbnMatch || !ZxcvbnFreeInfo ){
23   //error: Any of the two functions was not resolved correctly
24 else{
25   //we can use the functions normally
26
27 //When not needed any more
28 zxcvbnLib->unload();
29 ZxcvbnMatch = 0;
30 ZxcvbnFreeInfo = 0;
31 free(zxcvbnLib);
```

zxcvbn algorithm broke down the password. The user can see this information in a table like the one in figure 3.19

3.4.9 PassPhrase Generator

The `PassPhraseGen` C++ function implements the PassPhrase Generator. The function call is done from the `AddEntry` class, when the `Generate`

Listing 3.22: ZxcvbnMatch function declaration

```
1 double ZxcvbnMatch(        //Returns: entropy value in bits.
2
3   const char *Passwd,      //The password to be tested. Null
       terminated string.
4
5   const char *UserDict[], //User supplied dictionary words to be
       considered particulary bad. Passed as a pointer to array
       of string pointers, with null last entry (like the argv
       parameter to main()). May be null or point to empty array
       when there are no user dictionary words.
6
7   ZxcMatch_t **Info        //The address of a pointer variable to
       receive information on the parts of the password. This
       parameter can be null if no information is wanted. The data
       should be freed by calling ZxcvbnFreeInfo().
8 );
```

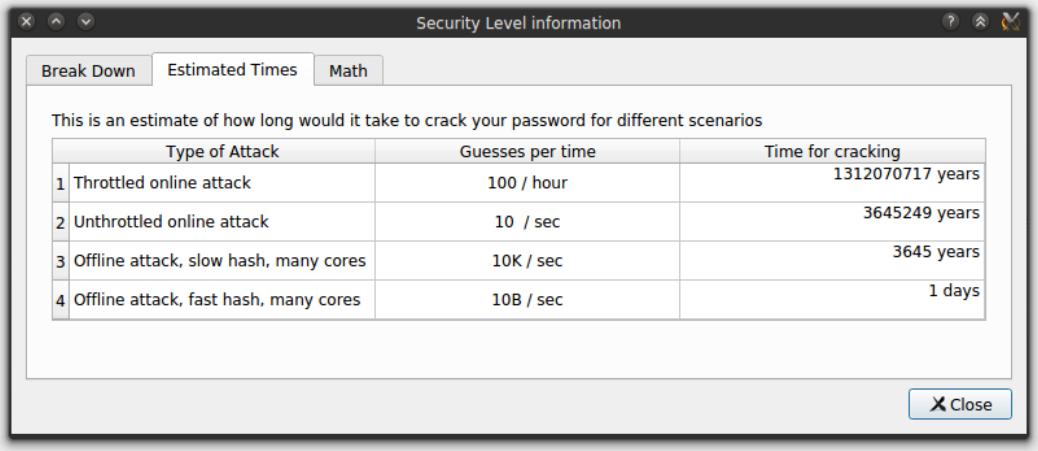


Figure 3.18: Crack times for different attacker capabilities

`Password` button is clicked.

`AddEntry` reads the configuration values stored as `QSettings`, asserts them and then makes the call. This values can be modified by the user in the preferences window, shown in figure 3.20. After the user selects the dictionaries and tunes the available options, they must click the apply button,

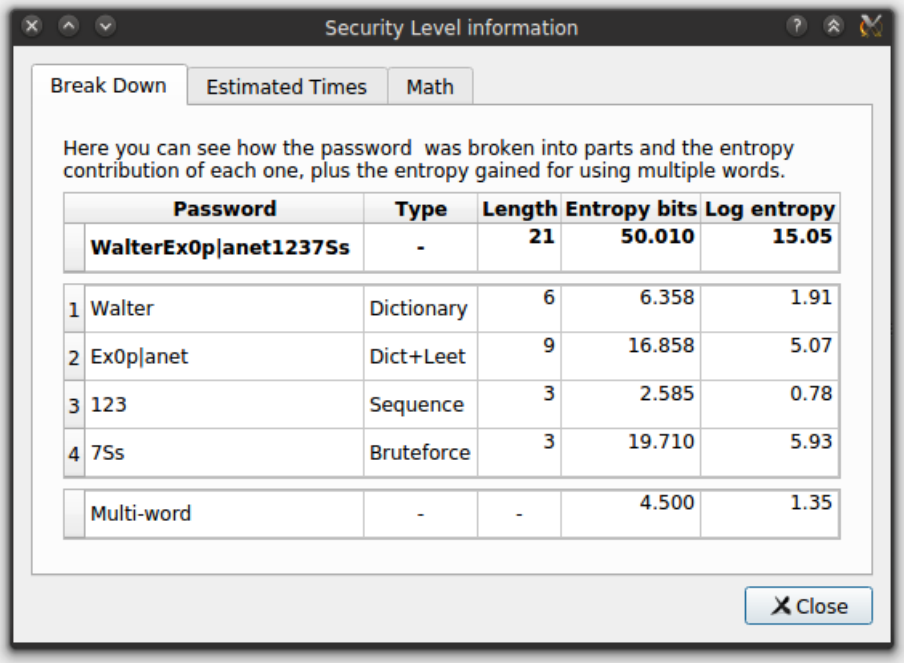


Figure 3.19: Password broke down by the zxcvbn algorithm

which will trigger a line by line read of the dictionaries, to count the number of lines, that is the number of words available. This is necessary as it is not possible to know how many lines a file has without counting them, and the total number is required in order to generate properly bounded random numbers in the PassPhraseGen function.

The function declaration is shown in listings 3.23

The function first generates `numWords` random numbers in the range `[1, totalLen]` with the Qt function `QRandomGenerator` (introduced in version 5.10). The generated numbers represent *Line numbers* in the dictionary files. As each line contains a word, the function indeed extracts random words. There are a few things to consider in this process:

1. It is impossible to read a random line from a TextFile without reading all the previous lines first. So, in order to extract the words it is necessary to read the dictionary line by line and keep a counter to know the number line we are at. To speed up the process making sure each line is read only once, the random numbers are sorted in ascending order first. The dictionary can then be read line by line extracting the words where the

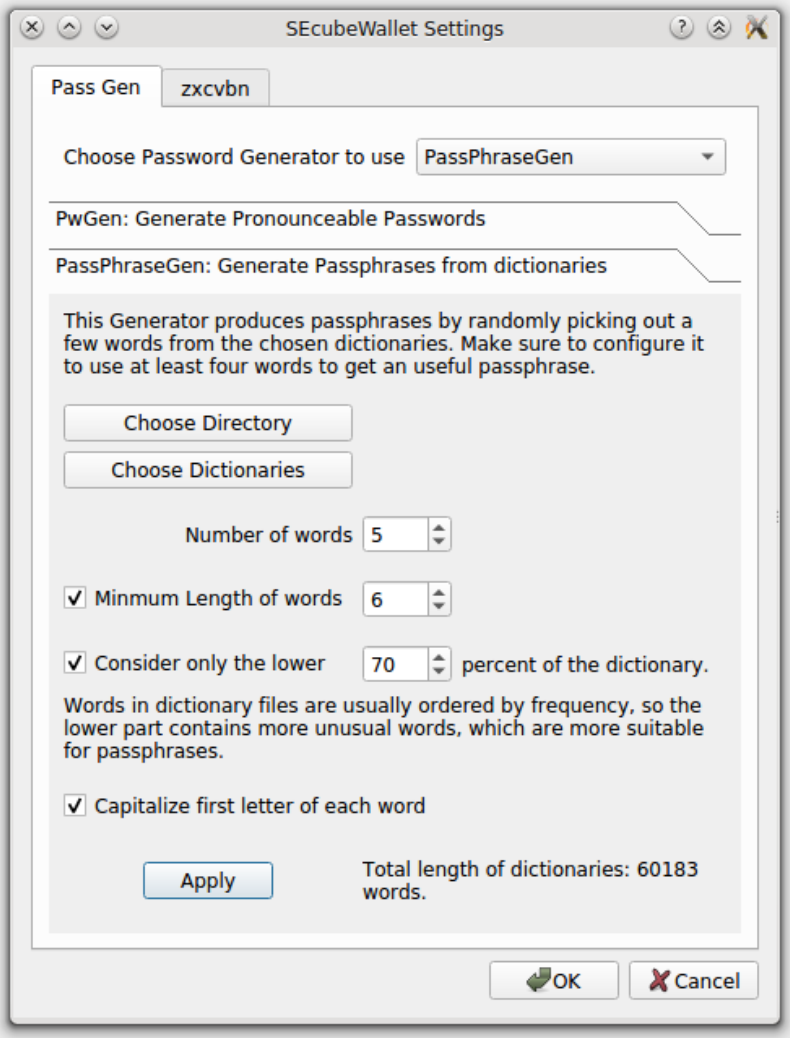


Figure 3.20: Settings for PassPhrase Generator

counter equals one of the random numbers. When the last random line is extracted, the file can be closed.

2. There can be multiple dictionaries. The generated random numbers span the total of available words, so some random lines will be in some dictionaries, some in others. Therefore, using the `dictsLen` list, which contains the cumulative lengths of the dictionaries, we need to determine in which dictionary and in which internal line, each random number is. With this information, and the random numbers ordered, it is possible to

Listing 3.23: PassPhraseGen function declaration

```
QString PassPhraseGen(  //Return: Generated PassPhrase
  QString path,         //path to dicts
  QStringList dicts,    //list of dicts
  QStringList dictsLen, //cumulative list of dics lengths
  int totalLen,         //total number of candidate words
  int numWords,         //number of words in the password
  bool ppgenMinLenEnab, //use only words longer than min len
  int ppgenMinLen,      //min length
  bool capFirst,        //uppercase first letter of each word
  bool ppgenLowerEnab,  //use only lower part of dicts
  int ppgenLower        //how much of the lower part to use
);
```

extract the words efficiently and making sure dictionary files are opened only if necessary and only once.

3. When the minimum length option is enabled, the total number of available words is reduced. The algorithm accounts for this fact by counting only the lines with a word larger than the minimum length. This is done both in the preference window, where the total number of lines is counted, and at the word extraction process. In this way, the random lines are pinpointed as before, by reading line by line and comparing the counter; the working logic is not altered, it just ignores the "disabled" short lines.
4. Finally, to consider only the lower part of each dictionary the preferences window counting process is not altered, and the modifications are all done at the extraction process. If for example, the user wants to work with the lower 30%, the random generated numbers are now bounded to `[1, (0.3)totalLen]`. The corresponding dictionary and internal line for each random number are calculated by taking into account that the first 70% of each dictionary must be skipped. With this two values, the files can be read as in the simple case. As this process is different from the minimum length one, they do not interfere with each other.

Table 3.3 presents some PassPhrases examples for different configurations, along side the score given by the zxcvbn estimator. The estimator uses the same dictionaries as the generator, so this assumes a worst case scenario where the hacker has access to all the possible words the user considered when creating the PassPhrase.

Two dictionaries where used: `words-eng_wiki.txt` with 100000 lines and `words-it_badip.txt` with 15480 lines (around 6 times smaller), so most of the extracted words will be English.

Table 3.3: PassPhrases examples for different configurations

PassPhrase	**No. of words**	**Min. word Len**	**% of dict. used**	**Log Entropy & Level**
Cocchio	1	-	-	4.27 (L1)
Melun	1	-	-	4.93 (L1)
Legitimately	1	8	-	4.55 (L1)
Woodhaven	1	8	30%	4.94 (L1)
VestaOrman	2	-	-	7.78 (L2)
ShorelineCech	2	-	-	9.18 (L3)
MongoliaSimpsons	2	8	-	7.30 (L2)
McinnisPhaya	2	-	30%	9.14 (L3)
ZucchiniSalamandra	2	8	30%	9.19 (L3)
SacchettiVigevano	2	8	30%	9.11 (L3)
DrammaturgicoSbatacchiare	2	12	-	8.98 (L3)
MalformationsAstrophysical	2	12	-	9.60 (L3)
LatinaInterchangeFbo	3	-	-	13.5 (L4)
OsaAymanCantinflas	3	-	-	12.98 (L4)
ImmobileCwSites	3	-	-	11.43 (L4)
RimmelBragFaenza	3	-	30%	13.49 (L4)
RecliningCanberraEcuadorian	3	8	-	13.69 (L4)
SeashellsHippocraticCameroun	3	8	30%	14.90 (L4)
InaspettatoRothschildsDisconcerting	3	8	30%	14.48 (L4)

The results in the table indicate the most important parameter is the

number of words. Three words are enough to reach a zxcvbn level 4, which, as seen in previous sections, is very secure. A two word PassPhrase as long as `DrammaturgicoSbatacchiare` is not better than the shorter `ImmobileCwSites`, because the latter has one more word. The minimum word length also influences the PassPhrase entropy, but their effects are not as pronounced. Finally, working only with to the lower part of the dictionary may seem to not have any effect, but in reality its use is crucial: it ensures the generated PassPhrases do not contain any of the most common words, like `the` or `essere`. From the results on the table we can not appreciate this fact, but at least we learn the option does not do any harm either. (Although if the attacker finds out the user is generating PassPhrases using only the 30% most uncommon words on a given language, their job would get easier).

3.4.10 The FAT32 bug

During the application development a strange and elusive bug was found: The `secureSQLite` library did not work properly with the FAT32 file system. When the save directory was in a FAT32 system, (either a partition, an external hard disk or an USB stick), the created databases were not stored properly, and when trying to open them they were completely empty. Only the database name was saved; any table in the wallet was lost. Moreover, after saving, besides the `.sqlite` file, a `.sqlite-journal` was created. Figure 3.21 depicts the presences of this file in a FAT32 file system.

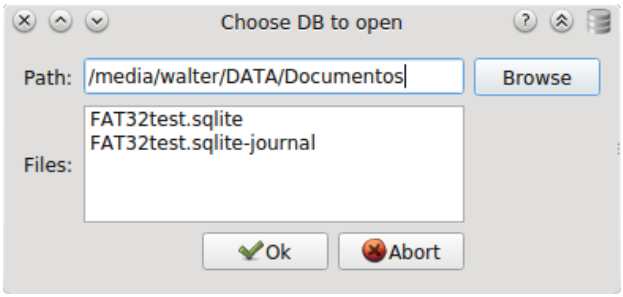


Figure 3.21: secureSQLite Databases in a FAT32 file system

This file is known as a rollback journal and is used by the SQLite standard to avoid database corruption when a transaction can not be completed. From the SQLite official documentation [15]: “The rollback journal is always located in the same directory as the database file and has the same name as the database file except with the 8 characters "-journal" appended. The rollback journal is usually created when a transaction is first started and is usually

deleted when a transaction commits or rolls back. The rollback journal file is essential for implementing the atomic commit and rollback capabilities of SQLite. Without a rollback journal, SQLite would be unable to rollback an incomplete transaction, and if a crash or power loss occurred in the middle of a transaction the entire database would likely go corrupt without a rollback journal. The rollback journal is usually created and destroyed at the start and end of a transaction, respectively. But there are exceptions to this rule. If a crash or power loss occurs in the middle of a transaction, then the rollback journal file is left on disk. The next time another application attempts to open the database file, it notices the presence of the abandoned rollback journal (we call it a "hot journal" in this circumstance) and uses the information in the journal to restore the database to its state prior to the start of the incomplete transaction."

The presence of the journal file hints at the problem being an interruption during the execution of `secureSQLite` write functions when working on a FAT32 system. A few test were performed to try to narrow down the error origin:

- The application was tested in other computer running a different OS, and the problem persisted, always in FAT32 systems, meaning it was not an OS specific issue.
- The application `secureSQLiteBrowser` which also uses the secureSQLite library was tested and presented the same problem, meaning it was not an error in the library usage.
- The demo applications `SEfile_IMG` and `SEfile_TXT` were also tested, and worked without any problems in FAT32 partitions. These applications do not use `secureSQLite`; they leverage directly the `SEfile` library, which is also used by the `secureSQLite` library, meaning the problem was not in `SEfile` but in `secureSQLite`. This was proved to be a wrong assumption later on.

With these points in mind, the bug was searched inside the `secureSQLite` library, but after several debug sessions, it was possible to trace the error origin to the `SEfile` library. Specifically, in the file `SEfile.c`, and function `secure_seek`. Figure 3.22 shows the actual traceback, from the call to `sqlite3_exec` in the `Save_Wallet` action, all the way up to the `secure_seek` function.

The actual error is produced in the call to the OS function `lseek()` in line 18 of listing 3.24.

Level	Function	File	Line
1	secure_seek	SEfile.c	988
2	SecureRead	sqlite3.c	36936
3	sqlite3OsRead	sqlite3.c	19560
4	syncJournal	sqlite3.c	50796
5	sqlite3PagerCommitPhaseOne	sqlite3.c	52927
6	sqlite3BtreeCommitPhaseOne	sqlite3.c	62209
7	vdbeCommit	sqlite3.c	72936
8	sqlite3VdbeHalt	sqlite3.c	73339
9	sqlite3VdbeExec	sqlite3.c	78420
10	sqlite3Step	sqlite3.c	75878
11	sqlite3_step	sqlite3.c	75939
12	sqlite3_exec	sqlite3.c	108869
13	MainWindow::on action Save Wallet triggered	mainwindow.cpp	829

Figure 3.22: Error Traceback

The `lseek()` function, as defined in the linux manual page [7]:

```
off_t lseek(int fd, off_t offset, int whence);
```

"**`lseek()`** repositions the file offset of the open file description associated with the file descriptor `fd` to the argument `offset` according to the directive `whence` as follows:

`SEEK_SET` The file offset is set to offset bytes.

`SEEK_CUR` The file offset is set to its current location plus offset bytes.

`SEEK_END` The file offset is set to the size of the file plus offset bytes. lseek() allows the file offset to be set beyond the end of the file (but this does not change the size of the file)

Return Value: Upon successful completion, `lseek()` returns the resulting offset location as measured in bytes from the beginning of the file. On error, the value `(off_t) -1` is returned and `errno` is set to indicate the error."

`secure_seek()` can be seen as a wrapper that uses the OS function `lseek()` to move the file pointer correctly, taking into account the way the `SEfile` library redefines a file, including the overhead added by sectors and headers that allow the data to be encrypted.

To better understand what was causing the error, the `return` and `errno` values for two save operations of the same database were compared. One into a FAT32 file system and the other into a ext4 system.

When the FAT32 system was used, the `return` value was indeed `(off_t)-1`, and the `errno`, was set to `22`, which corresponds to:

"`EINVAL`: `whence` is not valid. Or: the resulting file offset would be negative, or beyond the end of a seekable device."

When saving into an ext4 file system, the `return` value was `9844` and

Listing 3.24: FAT32 Error origin at secure_seek

```
1  /**This function is used to move correctly the file pointer.
2   * [in]  hFile     The handle to the file to manipulate.
3   * [in]  offset    Amount of character we want to move.
4   * [out] position  Pointer to int32_t to store the final pos.
5   * [in]  whence    Move from file start, end or current pos.
6   * [return]        Returns '0' in case of success.*/
7  uint16_t secure_seek(SEFILE_FHANDLE *hFile, int32_t offset,
       int32_t *position, uint8_t whence){
8    ...
9    buffer_size = *position - file_length;
10   if(buffer_size>0){
11     //if destination exceed the end of file, empty sectors are
           inserted at the end to keep the file consistency
12     buffer=calloc(buffer_size, sizeof(uint8_t));
13     if(buffer==NULL)
14       return SEFILE_SEEK_ERROR;
15
16     if((file_length%SEFILE_LOGIC_DATA)){
17       errno = 0; //clear errno
18       hTmp->log_offset=lseek( // the error is produced here
19         hTmp->fd,
20         ((file_length%SEFILE_LOGIC_DATA)-SEFILE_SECTOR_SIZE),
21         SEEK_END);
22       error = errno;//capture errno, 22 when in FAT32 systems
23       if(hTmp->log_offset==4294967295)
24         //lseek returns (off_t)-1 when error. off_t is 32 bits,
               so 2^32-1 = 4294967295
25         return SEFILE_SEEK_ERROR; //the upcoming secure_write
               call fails anyway
26
27       if(secure_write(&hTmp, buffer, buffer_size)){
28         free(buffer);
29         return SEFILE_SEEK_ERROR;
30       }
```

`errno` was equal to zero.

Figure 3.23 shows this difference, and it is worth noticing that besides the file descriptor and pointers (always different for different executions), the rest of the variables have the same value

Going back to the `secure_seek()` code in listing 3.24, the line 18 moves

Name	Value	Type
absOffset	9844	int32_t
buffer	0	uint8_t
buffer_size	496	int32_t
dest	10374	int32_t
error	22	int
file_length	8720	uint32_t
▸ hFile	@0xb8f098	SEFILE_FHANDLE
▾ hTmp	@0xb932b0	SEFILE_FHANDLE
fd	19	int32_t
log_offset	4294967295	uint32_t
▸ nonce_ctr	@0xb932b8	uint8_t[16]
▸ nonce_pbkdf2	@0xb932c8	uint8_t[32]
offset	9216	int32_t
overhead	646	int32_t
position	9216	int32_t
sectOffset	116	int32_t
tmp	0	int32_t
whence	0	uint8_t

(a) FAT32 filesystem

Name	Value	Type
absOffset	9844	int32_t
buffer	0	uint8_t
buffer_size	496	int32_t
dest	10374	int32_t
error	0	int
file_length	8720	uint32_t
▸ hFile	@0xd2e838	SEFILE_FHANDLE
▾ hTmp	@0xd2ad10	SEFILE_FHANDLE
fd	23	int32_t
log_offset	9844	uint32_t
▸ nonce_ctr	@0xd2ad18	uint8_t[16]
▸ nonce_pbkdf2	@0xd2ad28	uint8_t[32]
offset	9216	int32_t
overhead	646	int32_t
position	9216	int32_t
sectOffset	116	int32_t
tmp	0	int32_t
whence	0	uint8_t

(b) ext4 filesystem

Figure 3.23: Return and errno values for a save operation

the file pointer to the last `LOGIC_DATA` byte, because in order to keep the file consistency, its is required to write empty sectors starting from this position until the end of the file. To do so, starting from the end of the file (thus the argument `SEEK_END`), the function moves the pointer backwards the number of empty bytes in the last sector.

In order to move backwards, the `offset` argument of `lseek()` must be negative. In this case, the argument is equal to:

```
offset = (file_length%SEFILE_LOGIC_DATA)-SEFILE_SECTOR_SIZE
```

Which should always be always negative, but it turns out, in the case of the FAT32 file system, this offset is not interpreted as a 2's complement negative value, but as a very large positive one. Therefore, the `lseek()` function tries to move the pointer outside the file limit, explaining the error: `EINVAL`: the resulting file offset would be negative, or beyond the end of a seekable device.

To fix the error, a simple cast to `int32_t` for the `offset` argument is enough, resulting in the call:

```
hTmp->log_offset=lseek(
  hTmp->fd,
 (int32_t)((file_lengh%SEFILE_LOGIC_DATA)-SEFILE_SECTOR_SIZE),
  SEEK_END);
```

This kind of casting was actually found in other function calls in the

`SEfile.c` file, so it made sense to use it in this situation too.

The reason the problem only affected FAT32 partitions is probably related to the fact that, as `lseek()` needs to move a file pointer, low-abstraction level functions need to be used, which are have different implementations for each type of file system. The ext4 low level functions were able to interpret the offset as negative whereas the FAT32 took it as positive.

Why the error only appears when using the `secureSQLite` library, and not in the demo applications `SEfile_IMG` and `SEfile_TXT` honestly remains a mystery. It may be that the condition in line 10 of listing 3.24 (`if(buffer_size>0)`), that evaluates if the destination exceeds the end of file, is only met in rare occasions, and the way SQLite journaling works is one of them.

Chapter 4

Results, Discussion and Future work

4.1 SEcubeWallet weaknesses

In this section, the weaknesses of the SEcubeWallet application are discussed, as well as some ideas about how to fix them.

4.1.1 First table corruption

As explained in session 3.4.3, during the application development it was impossible to find the origin (and solution) to an error concerning the database saving and opening process: the first table in a wallet is always corrupt. The problem is not present when using the regular SQLite API; it only appears when using the secured version. This seems to point out the error is the secureSQLite library's fault. But because the secureSQLiteBrowser application does not have this problem it is not possible to discard SEcubeWallet as the origin of the problem.

The secureSQLiteBrowser and SEcubeWallet applications differ in the way they use the secureSQLite API. secureSQLiteBrowser, as it is a data base manager, uses complex SQLite functionalities like save points and pragma statements to exploit all of the API's capabilities, while SEcubeWallet only uses simple open/exec/close SQLite functions. So in a way SEcubeWallet is responsible for the error that could be avoided by using more complex SQLite statements. But the secureSQLite library should behave as the regular SQLite API, specially in the most simple cases.

In any case, the implemented workaround in SEcubeWallet (always having an unused and empty first table to sacrifice) should be considered temporary, not only because it is not optimal, but more importantly, because the unfixed error may cause other problems in the future.

4.1.2 The FAT32 bug

Even if the FAT32 bug (see section 3.4.10) can be considered to be fixed, the truth is its origin is not completely clear, and it may be the case the adopted solution (casting to int) only works for a subset of cases.

4.1.3 Only Linux has been tested

The SEcubeWallet application has only been tested running on Linux. Although Qt is platform independent, some of the application functionalities may need some changes to work on Windows or Mac systems. For example, the zxcvbn compilation process relies on gcc commands, that may not be available in Windows. Moreover the FAT32 bug is a very OS specific issue, as it involves the use of the lseek() function (Linux and Mac). It may be the case the bug does not exists in a Windows machine, or on the contrary there are other errors waiting to be fixed.

4.1.4 Missing icons

The reader of this thesis may have noticed the lack of icons in all of the SEcubeWallet Windows. Due to tight time constraints, it was decided to allocate most of the efforts into other aspects of the application, but that is not to say icons are not important. Future version of the application will definitely have icons to increase the user experience.

4.2 Future work

In this section, a few ideas about how to extend the SEcubeWallet functionalities are given. None of them were considered for this work as they are either too time consuming to implement, or are out of the author's expertise.

4.2.1 SEkey integration

SEkey is a new library currently under development at Politecnico di Torino by Mateus Françani as his master thesis work. The library will sit next to SEfile and SElink, as depicted in Figure 4.1 taken from [23].

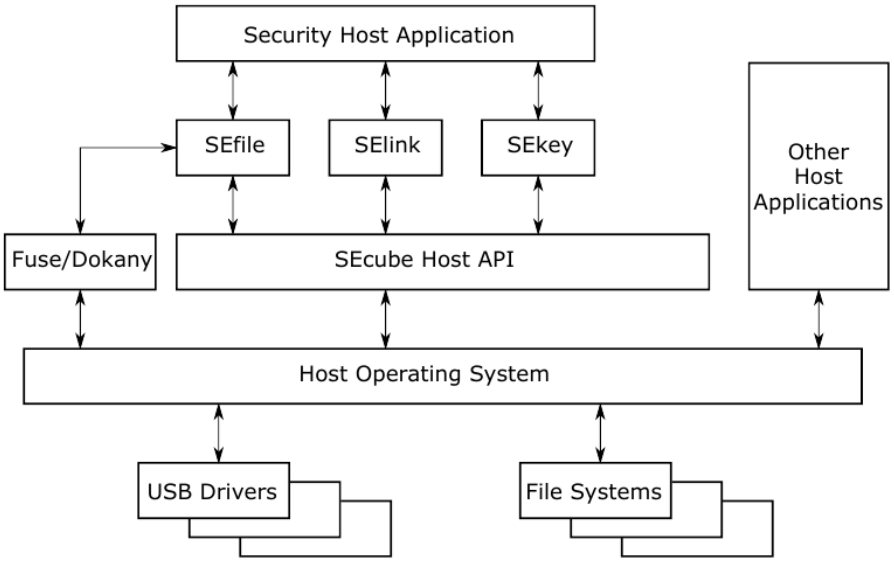


Figure 4.1: Host side SEcube™ architecture, including the SEkey library

The library will work as a key management system for the SEcube™ framework. Right now keys inside a SEcube™ chip can only be modified at factory reset. This is not very useful in a working environment, as the purpose of having multiple keys is to allow users to share information with selected people (people sharing a key are known as a group). The job of the SEkey library will be to allow an administrator to dynamically add and remove keys to SEcube™ devices using an intuitive GUI. By doing so, the administrator can conform groups of users, that later on can use their SEcube™ devices to share sensitive information, knowing it will be secured against unauthorized access (from people outside the group).

As the SEcubeWallet application already offers an intuitive GUI for the management of passwords, it could be extended to support the management of keys as well. If the person login in is an administrator, the application should offer the possibility to edit the keys present in the SEcube™ device. If it is a regular user, it should only let them see what keys they can use, i.e. to which groups they belong.

4.2.2 Browser integration

The vast majority of users store their internet passwords within their preferred web browser, alongside other sensitive information like browsing history and bookmarks. This allows them to use their passwords easily and fast, as they can for example autofill login credentials in websites.

Exploit the capabilities of the SEcube™ platform to protect all that information while maintaining ease of use and transparency to the final user would be a great advance in the purpose of reaching as many users as possible.

A couple of alternatives come to mind in order to implement this integration:

- Borrow the idea followed by the Mooltipass system, porting the entire Qt application to a complement for the most popular web browsers (Firefox, Chrome, Opera).
- Implement a web browser complement that "talks" with the SecubeWallet application and request for passwords when the user wishes to autofill a login field.
- Follow the auto-type approach used by the software password manager KeePass. [2]. "KeePass features an "Auto-Type" functionality. This feature allows you to define a sequence of keypresses, which KeePass can automatically perform for you. The simulated keypresses can be sent to any other currently open window of your choice (browser windows, login dialogs, ...). By default, the sent keystroke sequence is {USERNAME} {TAB} {PASSWORD} {ENTER}, i.e. it first types the user name of the selected entry, then presses the Tab key, then types the password of the entry and finally presses the Enter key."

In either case ensuring the new functionalities do not compromise the security of the passwords must be the top priority.

4.2.3 Mobile application (Android)

To have a functional SEcubeWallet for the Android system, three elements would need to be ported:

- The SEcube™ chip. As far as the author knows, the Blu5 group is working on a product that would allow to use the SEcube™ hardware platform in smartphones running the Android OS.

- The software libraries. The SEfile and secureSQLite libraries are written to work on Linux, Windows and Mac systems. Given that Android is based on Linux, porting the libraries should be possible.
- The GUI. Qt for Android [13] "allows to run Qt 5 applications on devices with Android v4.1 (API level 16) or later". All of the Qt modules used by SEcubeWallet are supported, but a GUI redesign is required in order to cope with the constraints imposed by a smaller display.

4.2.4 Custom columns

Wallets in the SEcubeWallet application have a fixed set of columns (USERNAME, DOMAIN, PASSWORD, DESCRIPTION, DATE). This set should be enough in most of the cases as they are the standard for wallet managers, but some users may want to add other custom columns, for instance to store "security questions and answers" used by some websites, an email associated to the account, or multiple passwords in the same entry.

4.2.5 Expired passwords notification

Changing passwords regularly is important to keep a high level of security in any system. If the user has a large number of passwords it may be hard to remember when one of them needs to be updated. SEcubeWallet already offers the possibility to search for passwords older than a given period of time (for example passwords older than six months). This functionality could be extended to notify the user when a password needs to be updated. The user could configure the expiration time for each password individually, or for all of them, and how they wish to be notified (Message inside the application, desktop notification, email, etc)

Bibliography

[1] About Qt. https://wiki.qt.io/About_Qt.

[2] Auto-Type. https://keepass.info/help/base/autotype.html.

[3] Banca dati dell'italiano parlato . http://badip.uni-graz.at/en/corpus-lip/list-of-lemmata.

[4] Blue5 Group. http://www.blu5group.com/.

[5] Column Aligned Layout. https://github.com/sashoalm/ColumnAlignedLayout.

[6] DB Browser for SQLite. http://sqlitebrowser.org/.

[7] Linux Programmer's Manual: LSEEK. http://man7.org/linux/man-pages/man2/lseek.2.html.

[8] Model/View Programming. http://doc.qt.io/qt-5/model-view-programming.html.

[9] Mooltipass. https://www.themooltipass.com/.

[10] Mooltipass Github repository. https://github.com/limpkin/mooltipass.

[11] Mooltipass hackaday page. https://hackaday.io/project/86-mooltipass-offline-password-keeper.

[12] Qt Creator. http://doc.qt.io/qtcreator/.

[13] Qt for Android. http://doc.qt.io/qt-5/android-support.html.

[14] SEcube SDK. https://www.secube.eu/resources/.

[15] Temporary Files Used By SQLite. https://www.sqlite.org/tempfiles.html.

[16] Wiktionary:Frequency lists. https://en.wiktionary.org/wiki/Wiktionary:Frequency_lists.

[17] zxcvbn: Low-budget password strength estimation, c/c++, on github. https://github.com/dropbox/zxcvbn.

[18] zxcvbn: Realistic password strength estimation.

https://blogs.dropbox.com/tech/2012/04/zxcvbn-realistic-password-strength-estimation/.

[19] *SEcube Data Sheet introduction*, 2015.

[20] Airo, G., Prinetto, P., Carelli, A., Somma, G., and Varriale, A. *SEcube Development Kit: Getting Started*, 2017.

[21] Airo, G., Prinetto, P., Ferri, N., Carelli, A., Scalia, G., Somma, G., and Varriale, A. *SEcube Development Kit: L2 User Manual*. 2017.

[22] Burnett, M. 10,000 Top Passwords. https://xato.net/10-000-top-passwords-6d6380716fe0.

[23] Farulla, G. A., Pane, A. J., Prinetto, P., and Varriale, A. An object-oriented open software architecture for security applications. In *2017 IEEE East-West Design Test Symposium (EWDTS)* (Sept 2017), pp. 1–6. https://ieeexplore.ieee.org/document/8110070/.

[24] Stockley, M. Why you STILL can't trust password strength meters. https://nakedsecurity.sophos.com/2016/08/17/why-you-still-cant-trust-password-strength-meters/.

[25] Toponce, A. A document evaluating different open source password generators and password strength testers. https://gist.github.com/atoponce/173c113ea4a81a9657148ce5d4fa2fd3.

[26] Ts'o, T. pwgen(1) - Linux man page. https://linux.die.net/man/1/pwgen.

[27] Wheeler, D. L. zxcvbn: Low-budget password strength estimation. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 157–173. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/wheeler.

[28] xkcd. Password strength. https://xkcd.com/936/?