Python password manager documentation

This document provides a brief description of the software manager and its components as well as aims to explain why and how things are implemented the way they are.

Requirements

`Python 3.11.7` (may work on earlier versions)

The following packages that are also found in requirements.txt:

```
argon2-cffi==23.1.0

pyperclip==1.8.2

cryptography==42.0.2
```

They can be installed from the directory using `pip install -r requirements.txt`

Features

The program provides a simple way to store and manage passwords locally. This password manager is suitable for multiple users. The user must register themselves before they can add passwords to their services. After creating a user and logging in they can start saving passwords to the services. It supports multiple users for example if multiple family members use the computer and use different services.

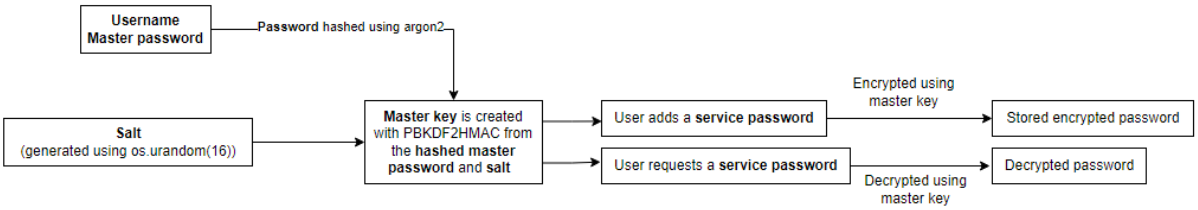
-Before logging in

- -Add users
- -Log in
- -Exit

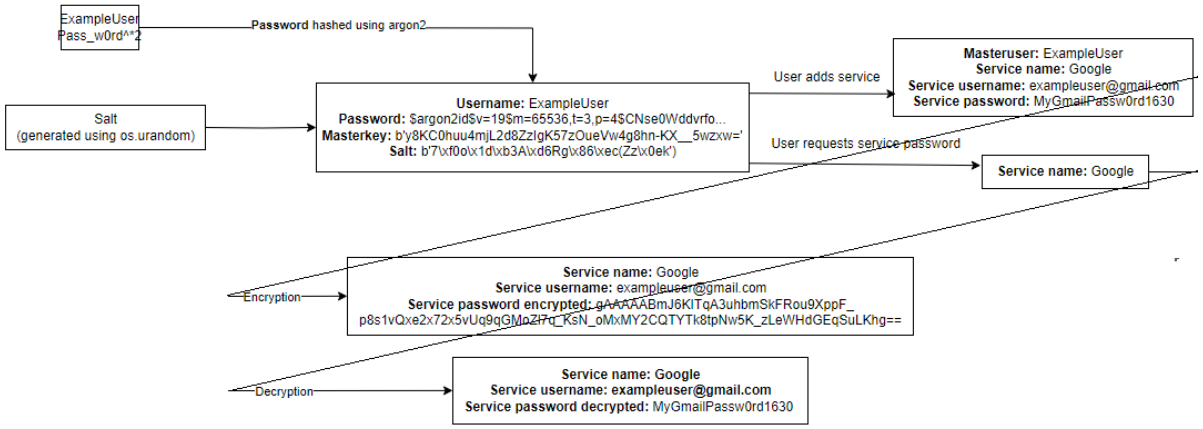
-After logging in the user can handle their own information

- -Add services
- -Delete services
- -View all services, but not their passwords
- -Get service passwords
- -Delete their user and all services associated with it
- -Log out
- -Exit

Below is a simplified example of a common use case. The image does not cover creating user, deleting services etc. and more details about the algorithms can be found below



Below is the same example with examples of real values.



Database

The software utilizes a database that has two tables that are created using SQLite in the following way:

```
    cur.execute("""CREATE TABLE users (
            ID INTEGER PRIMARY KEY,
            username TEXT NOT NULL UNIQUE,
            masterpassword TEXT NOT NULL,
            masterkey TEXT NOT NULL,
            salt TEXT NOT NULL
            );""")

    cur.execute("""CREATE TABLE servicepasswords (
            masteruser TEXT NOT NULL,
            servicename TEXT NOT NULL,
            serviceuser TEXT NOT NULL,
            password TEXT NOT NULL
```

```
        );""")
```

In the database's *users* table ID is simply a unique number. Username is chosen by the user and there are no limitations on it other than that it has to be 1-24 characters long. The salt is generated automatically. The master key is also generated automatically from the user's password and salt. Further details about these are provided below.

The servicepasswords table contains the same username as the users table, but it's renamed to masteruser to differentiate from the service's username. Service name contains the name of the service (e.g. Google). Serviceuser contains the username for that service (e.g. testuser@gmail.com) and the password field contains the encrypted password for that service. No limitations are enforced for these fields other than that the service name must be at least one character long.

Master password

The password has to be 8 characters or longer and has to be less than 64 characters long. This limitation is based on OWASP's authentication cheat sheet which references NIST (https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html). As per the recommendation all characters are allowed. When creating a password, the user also has an option to generate a password that follows good security practises. The generated password is a 16-character long string containing at least one lowercase character, uppercase character, a digit, and a punctuation character from Python's *string* library (https://docs.python.org/3/library/string.html). The randomness is ensured by using Python's *secrets* module that is recommended by OWASP's cryptographic storage cheat sheet. (https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html#secure-random-number-generation)

Salt

For PBKDF2HMAC to create a master key, a salt is generated using Python's os.urandom library. OWASP's cheat sheet doesn't mention any guidelines regarding salt length, but NIST does: "The length of the randomly-generated portion of the salt shall be at least 128 bits." (https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf) In the software this criteria is met by using os.urandom(16) to generate 128 bits of random data.

Os.urandom isn't directly endorsed by OWASP, but it is used for randomness in the *secrets* module that is recommended by OWASP's cryptographic storage cheat sheet (https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html#secure-random-number-generation). It can be found in the secrets module's source code for generating random bytes which is what we are using it for as well. (https://github.com/python/cpython/blob/3.8/Lib/secrets.py#L35)

Master key

The master key is generated from the salt and the master password. The master key is not stored in the database, but rather the components to create it and the user never has to use it themselves. An option would be to give the user the key in another format (e.g. a file), but this poses the risk of the user compromising the key themselves.

Service password

The service password has no limitations, nor can it have as the password practises and rules and requirements for a password vary from service to service. I.e. one service may require that the password is at least 6 characters whereas another service may require that the password is 8 characters and contains numbers and symbols.

Authentication

Authentication is implemented as a comparison of the Argon2id hashes. During authentication it's important to not tell information why it failed. For example, the software shouldn't tell "Authentication failed. Invalid password." or "Authentication failed. User does not exist." https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html

From OWASP: "Using any of the authentication mechanisms (login, password reset, or password recovery), an application must respond with a generic error message regardless of whether:

-The user ID or password was incorrect.

-The account does not exist.

-The account is locked or disabled."

SQlite3

"SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage." (https://docs.python.org/3/library/sqlite3.html)

SQLite is used for managing the database. It contains all the relevant credentials used in this software. The user cannot make custom SQL queries to this database.

Argon2id

Argon2 is a hashing algorithm that won the Password Hashing Competition in 2015. It was designed by Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Some use cases for it are hashing passwords and deriving keys. Due to the rising usage of GPUs and ASICs to crack passwords in parallel, Argon2 makes the computation of hashes "memory hard" (https://argon2-cffi.readthedocs.io/en/stable/argon2.html).

The reason I chose Argon2id is due to it being recommended by OWASP cheat sheet (https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html). I originally used bcrypt and then looked into scrypt, but OWASP recommended argon2 over bcrypt and scrypt. Argon2id in the password manager is ran with the default settings: *argon2.PasswordHasher(time_cost=3, memory_cost=65536, parallelism=4, hash_len=32, salt_len=16, encoding='utf-8', type=Type.ID).* In this software Argon2id is used to hash the passwords as well as in authentication (i.e. comparing hashes).

Fernet

Fernet is a python library used for symmetric encryption. Fernet utilizes AES 128 in CBC mode for encryption and decryption. (https://github.com/fernet/spec/blob/master/Spec.md) Fernet can be used to generate a key, or passwords can be used to derive a key. (https://github.com/pyca/cryptography/blob/main/docs/fernet.rst#using-passwords-with-fernet) The latter is being used to in this software. A key is being derived from the master password and salt that is then used with Fernet for encrypting and decrypting the service passwords.

PKDF2HMAC

PKDF2 stands for Password-Based Key Derivation Function 2 and HMAC means it utilizes a hash-based message authentication code. HMAC is used for verifying the integrity and authenticity of the message (or in this case a password).

PBKDF2 is mentioned in OWASP's cheat sheet as well as recommended by The National Institute of Standards and Technology (NIST): "Since PBKDF2 is recommended by NIST and has FIPS-140 validated implementations, so it should be the preferred algorithm when these are required. HMAC-SHA-256 is widely supported and is recommended by NIST." (https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#pbkdf2).

The password manager utilizes PBKDF2HMAC with the following configuration: PBKDF2HMAC(algorithm=hashes.SHA256(), length=32, salt=salt, iterations=600000) which aligns with the OWASP recommendation "PBKDF2-HMAC-SHA256: 600,000 iterations." In this software it used to derive valid keys for encrypting and decrypting service credentials.