Symmetric encryption using Fernet in Python - Master password use case

Asked 1 year, 6 months ago Active 1 year, 6 months ago Viewed 2k times

5

2

I've been trying to understand how symmetric encryption works and how I can integrate it in my CLI application but I've got stuck at some point which I'm going to describe below.

My use case is the following:

- I have a CLI application (`SQLAlchemy + click + Python 3.8`) which is going to be a very simple Password Manager (personal use).
- When started, I want to ask the user for a master password in order for him to be able to retrieve any information from a DB. If the user doesn't have a master password yet, I'll ask him to create one. I want all the data to be encrypted with the same master key.

To do all the above, I thought symmetric encryption would be the most suitable and Fernet came to mind, so I started writing some code:

```
import base64

from cryptography.fernet import Fernet, InvalidToken
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC


def generate_key_derivation(salt, master_password):
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
        backend=default_backend()
    )
    key = base64.urlsafe_b64encode(kdf.derive(master_password.encode()))
    return key


def encrypt(key, value_to_encrypt):
    f = Fernet(key)
    encrypted_key = f.encrypt(value_to_encrypt.encode())
    return encrypted_key


def decrypt(key, encrypted_key):
    f = Fernet(key)
    try:
        return f.decrypt(encrypted_key)
```

Now, I kinda tried to understand from the docs this:

> In this scheme, **the salt has to be stored in a retrievable location** in order to derive the same key from the password in the future.

Which, in my head means: store the salt in DB, and use it every time the user tries to use the application. Then, run the master password the user inserted through a key derivation function and check if it matches ... the key? **But I don't have the initial key since I didn't store it the first time along with the salt. And if I were to save it, wouldn't anyone be able to just use it freely to encrypt and decrypt the data?**

What's a common solution used to prevent the above?

Here is a small POC using `click`:

```
import os
import click

from models import MasterPasswordModel


@click.group(help="Simple CLI Password Manager for personal use")
@click.pass_context
def main(ctx):
    # if the user hasn't stored any master password yet,
    # create a new one
    if MasterPasswordModel.is_empty():

        # ask user for a new master password
        master_password = click.prompt(
            'Please enter your new master password: ',
            hide_input=True
        )

        # generate the salt
        salt = os.urandom(16)

        # generate key_derivation
        # this isn't stored because if it does anyone would be able
        # to access any data
        key = generate_key_derivation(salt, master_password)

        # store the salt to the DB
        MasterPasswordModel.create(salt)

    # if the user stored a master password, check if it's valid and
    # allow him to do other actions
    else:
        # ask user for existing master password
        master_password = click.prompt(
            'Please enter your new master password: ',
            hide_input=True
```

```
salt = MasterPasswordModel.get_salt()

# generate key_derivation
key = generate_key_derivation(salt, master_password)

# At this point I don't know how to check whether the `key` is
# valid or not since I don't have anything to check it against.

# what am I missing?
```

I hope all of this makes sense. As a TL;DR I think the question would be: How can I safely store the key so I can retrieve it for further checks? Or is that even how the things should be done? What am I missing? I'm sure I'm misunderstanding some things :)

LE: As specified in one of the comments, it looks like there might me a solution but I'm still getting stuck somewhere along the process. In this answer it's specified that:

> If you're not doing this already, I'd also strongly recommend **not using the user-supplied key directly**, but instead first **passing it through a deliberately slow key derivation function such as PBKDF2**, bcrypt or scrypt. You should do this first, **before even trying to verify the correctness of the key, and immediately discard the original user-supplied key and use the derived key for everything** (both verification and actual en/decryption).

So, let's take for example everything step by step:

1) I am asked for a master password for the first time. It doesn't exist in DB so, obviously, I have to create & store it.

2) Along with the newly generated salt, I have to save a hash of the provided master password (for the sake of example I'll use SHA-256).

3) I now have a record containing the salt and hashed master password so I can proceed further with using the app. I now want to create a new record in DB, which is supposedly going to be encrypted using my *key*.

The question is... **what key**? If I were to apply what's written above, I'd have to use my `generate_key_derivation()` function using the salt and hashed master password from DB and use that for encryption/decryption. But, if I do this, won't anyone be able to just take the hash_key stored in DB, and use the same `generate_key_derivation` to do whatever he wants?

So, what am I missing?

Share Follow

edited May 27 '20 at 19:39

asked May 24 '20 at 11:53

Grajdeanu Alex
310 3 17

Why do you want to store the key? Given the same master password and same salt you can derive the same key. So all you need to store is the salt. – President James K. Polk May 24 '20 at 16:21

@PresidentJamesK.Polk I know. But then how do you verify that the master password is the correct one if you don't have the key? – Grajdeanu Alex May 24 '20 at 16:23

Ah, I see, that's a good question. I can think of some ad-hoc ideas, like generating 32 additional bytes from the KDF and xor-ing the 16 bytes halves together and storing that. But you should use a properly analysed method and I don't know any off the top of my head. – President James K. Polk May 24 '20 at 16:29

There is a good answer on the crypto stack exchange for this: crypto.stackexchange.com/questions/1507/... – rfkortekaas May 26 '20 at 18:42

@rfkortekaas if I got that right (the former method from the accepted answer), I should add a hashing function (perhaps SHA-512) which will basically hash the `master_password` that the user will input and store that hash next to the salt, is that correct? – Grajdeanu Alex May 26 '20 at 19:23

1 Answer

Active | Oldest | Votes

1

I'm not a crypto expert, but I think the idea is to store the salt and a hash of the derived key like so:

1. get master password for first time
2. generate a salt
3. derive a new key using the salt and master password
4. discard the master password
5. hash the derived key
6. store the salt and derived key in the db
7. use the derived key to encrypt the stored passwords

Later use the salt and hash to verify the derived key is authentic like so:

1. get master password
2. get salt and hash from db
3. derive a key using the salt and master password
4. discard the master password

7. if it doesn't match, exit

8. otherwise, use the derived key to decrypt the other passwords.

Share Follow

answered May 29 '20 at 5:56

RootTwo

3,953 1 9 13

Hmm, even in this case, what stops someone to directly get the derived key from DB and use it to decrypt the other passwords? – Grajdeanu Alex May 29 '20 at 8:35

@GrajdeanuAlex. the derived key is **not stored anywhere**. The *hash* is stored. You can't use the hash to produce the generated key, you can only produce the generated key from the master password. The hash is only there to verify the master password efficiently. So, what stops someone from getting the derived key is that they can't get something that is not there to get. – Martijn Pieters ♦ Jun 19 '20 at 7:58