

Neil Patel, Abigail Santiago, Ben Santiago

CSC 382: Intro to Information Security

Dr. Sakib Miaz

12 May 2021

Open Sesame: A Password Manager

Abstract

Open Sesame is a simple, open-source password manager that is modeled after the successes of commercial password managers, such as their security principles/protocols, engineering design, and features offered. Open Sesame boasts a graphical user interface (GUI) that offers three unauthenticated actions and four authenticated actions. Unauthenticated actions allow a user to create an account, sign in to an account, and sign out of an account. Authenticated actions are based on the persistent storage model of CRUD (create, read, update, delete) and allow a user to add, get, update, or delete a password. In tandem, these actions allow authenticated users to interface with a local MySQL database which stores data that has gone through sanitization and encryption.

Introduction

As the world becomes more and more digital, users may find themselves integrating almost every aspect of their life into online services. Assuming that the user is following security practices and has a different password for each service, this leads to one user having many passwords - too many to remember. This begs the question: how and where should a user store their various passwords? This is where password management steps in. A password manager is a platform that allows users to consolidate their passwords and securely store them.

Password managers usually consist of an interface that allows users to add, delete, remove, and change their passwords, which are then encrypted and sent to a database, which are

all protected through a master password set by the user. Well-supported password managers often have a plethora of features and user-experience improvements like a copy-to-clipboard button, random password generator, multi-factor authentication, network-hosted databases, browser extensions, and form autofill.

While we were unable to implement many of these additional features, Open Sesame boasts a graphical user interface (GUI) that offers three unauthenticated actions and four authenticated actions. Unauthenticated actions allow a user to create an account, sign in to an account, and sign out of an account. Authenticated actions are based on the persistent storage model of CRUD (create, read, update, delete) and allow a user to add, get, update, or delete a password. Together, these actions allow users to interface with a local database that stores data that has gone through some sort of encryption.

Encryption is a key detail in password management, as it is a procedure that obfuscates information from curious eyes. Encryption takes plaintext, such as the password “helloworld123,” and encodes it as ciphertext. The ciphertext is a representation of plaintext that obscures all useful information from it, rendering it useless to anyone who gains access to it. Open Sesame uses encryption when storing your passwords, ensuring that an attacker who gains access to them will have no sensitive information of yours. In addition to passwords, Open Sesame encrypts other sensitive information, such as derivation keys.

Finally, this encrypted data needs to be stored somewhere and somehow. Open Sesame leverages a locally hosted MySQL database to store, retrieve, update, and delete all of this information. MySQL is a relational database management system, which means that the information is stored in tables that have relationships with one another.

Motivation and Current Systems

Apart from being central to daily life for many people, passwords are also often the main, if not only, point of authentication for many applications, services, and systems. However, many users do not take their password security seriously, often using short, easy to guess passwords that are often reused for many different sites and applications. Coupled with the fact that malicious actors have also evolved alongside this digital landscape, creating robust and secure passwords and password protection/management protocols have become increasingly important. And while multi-factor authentication is becoming increasingly common, it still does not make recycling passwords or using slight variations a safe venture. Many users often do not feel the repercussions of this until being on the tail-end of a security breach or an attack like the COMB (Compilation of Many Breaches) data breach that was leaked on a hacking forum this past February, which is a well-organized and searchable database containing 3.2 billion unique (plaintext) email-password pairs for services such as Netflix, LinkedIn, and Bitcoin (*COMB*, 2021).

In order to maintain long, hard-to-guess passwords that are different for every service, therefore ensuring a high level of security, many users turn to password managers so they do not have to go through the hassle of remembering all of their passwords. There are many password managers on the market, all of which do the same central goal of maintaining your passwords but all are unique in how they operate and what they offer. For example, take two popular password managers: the Google Chrome password manager extension, and LastPass. Both of which are extensions that either come loaded into the Google Chrome web browser or that can be downloaded into the Google Chrome web browser. Both automate many of the steps that arise in the process of making and storing passwords, such as asking users if they want to save their

password for a particular service, inputting their saved password into the login field, and even generating “suggested passwords” that are often long passwords comprised of numbers and letters with no distinguishable patterns or structure. Both also provide a user interface locked behind a master password, where the user can see and manage all of the services and passwords they have saved.

This knowledge, as well as our personal experience from using different password managers, led to us asking: can we build one ourselves? Our curiosity and desire to implement the numerous security concepts we studied throughout the semester and create some sort of digital tool motivated us to create Open Sesame. Open Sesame is a simple, open-source password manager that is modeled after the successes of commercial password managers, such as their security principles/protocols, engineering design, and features offered.

Contributions

When we began to research, we turned to already-existing password managers. These, along with numerous forum discussions on naive approaches to cryptography and security, gave us the understanding that we used to build Open Sesame. 1Password is a commercially available password manager, which has published many of the details surrounding its security design and the reasoning behind these decisions (*1Password Security Design*, n.d.). Due to this fact, 1Password proved to be an indispensable resource when building Open Sesame. Another notable influence was blog posts written by security experts that warned about the dangers of relying on homemade password managers. These posts specifically warned of the complexity of implementing cryptography without a proper understanding of it, and they explained that a fatal flaw in many programmers is a false sense of confidence that they can build secure password

managers. This overconfidence can be summed up by Scheiner's Law, which states “anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can’t break.” (“*Schneier’s Law*” - *Schneier on Security*, n.d.). While designing an algorithm you personally can not break is easy, designing one that *everyone else* can not break is a task of much greater scope and difficulty. Another humbling source was the paper, “Secure Password Managers and ‘Military-Grade Encryption’ on Smartphones: Oh, Really?” by Belenko and Sklyarov. This paper surveyed the current (2011) popular password managers for mobile devices, pointing out significant weaknesses in all of them. With this new-gained humility, we were able to acknowledge that we do not possess an understanding of cryptography that guarantees security. With this in mind, we continued to design and build Open Sesame (perhaps just a bit) less naively than we would have initially.

A key idea we kept in mind is Kerckhoffs's principle, the idea that “a cryptosystem should be secure even if everything about the system, except the key, is public knowledge” (“Kerckhoffs’s Principle,” 2021). In other words, complete transparency and understanding of a cryptosystem should not render it useless or breakable, as long as the derivation key is kept secret. This is why 1Password is able to publish detailed reports on the exact encryption schemes used by their systems. Even with this knowledge released to the public, the system is secure enough to not be cracked. We wanted to emulate this in Open Sesame, which meant that the encryption keys we used were the most valuable pieces of information.

1Password’s white paper on their security design was highly influential to us. It provided us with a full scope of what a commercial, fully secure, and fully operational password manager might look like. Moreover, it gave us insight into the minutia of their decision-making process, at least in regards to security. This introduced us to the idea of slow hashing, a process used to

stretch out the process of key derivation. While this is inconsequential to a user and how they might interact with the app, this process drastically slows down password cracking attempts. The paper also details their encryption, providing us with an idea of the level of security we might be able to provide, and helped us understand how we might perform user authentication.

When it came to database design and implementation, we turned to the paper “On the Security of Password Manager Database Formats,”(Gasti & Rasmussen, 2012) as well as the blog post “How do password managers work”(How Do Password Managers Work and Store Your Data?, 2021) on cybernews.com, the former explains different database formats across various popular password managers and assesses their level of security based on an adversary and system model that they created while the latter focuses on a more general database structure overview. Databases for password managers are often stored in at least one of three different ways: on a USB stick, in the cloud, or on a local device. Usually, it boils down to in the cloud or locally, if not both. Both have their own security risks that come associated with them. For the cloud, users are often unaware of where exactly their information is being stored, and they also need wifi or internet access to access their passwords. Storing the information locally gets rid of these problems, but brings with it the problems of only accessing passwords from the device they are stored in as well as if someone gets access to your device they have all of your information. Many bigger, online password managers, such as the Google Chrome extension, opt for the cloud.

From this, we decided to store all of our information locally, for the main reason being that for right now Open Sesame is a low-scale personal project created by non-experts. Essentially, this decision reduces the surface area of our problem space, meaning it would be easier and more sensible for us to keep everything locally. The next hurdle was what type of

database to use. According to the paper, a big drawback to a lot of predefined database management systems is their lack of secrecy and integrity, as many times variables and fields have to be named to access them, but these names, if seen, can reveal a lot of information about the passwords and how the password manager works (*How Do Password Managers Work and Store Your Data?*, 2021). To combat this, larger companies often create and implement proprietary data objects to store their information. Once again, given that we are operating on a smaller scale, we decided to use a predetermined database management system. We utilized a MySQL database with two different tables: one for all of the user's information to be generated by the master password, and then one (protected by the master password) that holds entries containing the name of a service, the username for the service, and the password for the service. We decided to use MySQL because that is what we all had the most experience with and, given that encryption and many other aspects of this project were already going to be challenging to implement, we thought that it would be beneficial to the project to have a simple, straightforward database that we can all understand and parse through easily.

Finally, we decided that we would use the Python Cryptography and Fernet libraries to accomplish our encryption. We chose Cryptography because of the functionality it provided: it allowed us to implement simple and secure encryption, but also provided a “hazardous materials” family of functions that allowed us to experiment with more detailed, and, if implemented improperly, more dangerous encryption. Cryptography provides slow hashing functions, password-based key derivation, key-based key derivation, authenticated encryption, and key wrapping. Additionally, Fernet uses 128-bit AES in CBC mode with PKCS7 padding, and HMAC with SHA256 for authentication. Finally, the initialization vector is created using the random function from the os Python library.

User Manual and Demo

- Install and configure MySQL Server (at least v8) @ <https://dev.mysql.com/downloads/installer/>
- Clone the GitHub repo @ <https://github.com/nepatel/open-sesame.git>
- Install any missing dependencies (base64, cryptography, python-dotenv, Goody, mysql-connector-python, os) using pip3
- Run "manager.py"
- If valid .env is not found for a DB connection, prompt the user to generate one on CLI
- From the Actions side panel, select...
 - "setup" to create an account (required input of username, master password)
 - "login" to login into an account previously setup (required input of username, master password)
 - "logout" to logout of an account (no input required)
- From the Actions side panel, select...
 - [NOTE: These functions will only work if you successfully login!]
 - "add" to add a password (required input of service, username, password)
 - "get" to get a password (required input of service, username)
 - "update" to update a password (required input of service, username, password)
 - "delete" to delete a password (required input of service, username)
- Click...
 - "Start" to perform the action
 - "Cancel" to exit the application

- If...
 - successful, click "OK" and view output
 - unsuccessful, raise a GitHub issue detailing your usage and copy the error log
- To...
 - perform another action, click "Edit"
 - exit the application, click "Close"
 - [NOTE: Unless you explicitly 'logout', you will remain logged in!]

GUI

_____To minimize time on non-essential features like the GUI, parsing inputs, and packaging, our group decided to leverage Gooey. Gooey is a Python library that creates a GUI for Python scripts with a few lines of code or less. Gooey attaches to our main function with a one-line decorator and some additional configuration options like a description, default window size, and disabling the restart button.

```
# attach Gooey to our code
@Gooey(program_name='open-sesame', program_description='An open-source password manager sans Alibaba and the Forty Thieves',
|      | default_size=(550, 440), show_restart_button=False)
def main():
```

Figure 1. Decorator code that attaches Gooey

Next, we have to set up a “parser” which is essentially the primary callback function for the application.

```
parser = GoeyParser()                # main app
subs = parser.add_subparsers(dest='command')  # main "function" for app
```

Figure 2. Initialize the main parser

However, since Open Sesame has several distinct actions, this by itself does not work for us. Thus, we divide the parser into smaller, discrete callback functions for each of our actions, or “subparsers.”

```
# add "sub-functions" (sub-parsers) to the main "function" (parser)
setup_parser = subs.add_parser('setup')
login_parser = subs.add_parser('login')
add_parser = subs.add_parser('add')
get_parser = subs.add_parser('get')
update_parser = subs.add_parser('update')
delete_parser = subs.add_parser('delete')
logout_parser = subs.add_parser('logout')
```

Figure 3. Dividing parser into subparsers

Then, we attach fields for user-input, or “arguments,” to each of these subparsers so that we can prompt the user for necessary data like a service, username, and password.

```

# add arguments (input fields) for parsers (functions)
setup_group = setup_parser.add_argument_group("Setup an Account")
setup_group.add_argument("Account Username")
setup_group.add_argument("Master Password", widget="PasswordField")

login_group = login_parser.add_argument_group("Login to Account")
login_group.add_argument("Account Username")
login_group.add_argument("Master Password", widget="PasswordField")

add_group = add_parser.add_argument_group("Add Password")
add_group.add_argument("Service")
add_group.add_argument("Username")
add_group.add_argument("Password", widget="PasswordField")

get_group = get_parser.add_argument_group("Get Password")
get_group.add_argument("Service")
get_group.add_argument("Username")

update_group = update_parser.add_argument_group("Update Password")
update_group.add_argument("Service")
update_group.add_argument("Username")
update_group.add_argument("New Password", widget="PasswordField")

delete_group = delete_parser.add_argument_group("Delete Password")
delete_group.add_argument("Service")
delete_group.add_argument("Username")

```

Figure 4. Adding arguments to subparsers

Before finally running the app, we add some sort of handler which executes the action chosen by the user. For example, here is how we use the user-inputted arguments in tandem with Goocy's parsers to create a "decision tree" of sorts with if-elif-else conditional statements.

```
# handle subfunctions and their args
if cmd == 'setup':
    print("Creating account...")
    account_username = args['Account Username']
    #username = account_username.lower()
    master_password = args['Master Password']
    create_user(account_username, master_password)
    print("Account created!")
    print("You must still 'login.'")
    print()
```

Figure 5. Handling subparsers (for sake of brevity, we have only included the first if-block)

In the unabridged version of Open Sesame's code, we see that Goocy's ease of use comes with a cost. It does not scale well with complexity. Even with only seven distinct actions, Open Sesame quickly becomes a jumbled mess of conditionals that brings iterative development to a grinding halt. Nonetheless, since it automatically generates a working GUI with minimal configuration, Goocy remains an excellent option for prototyping and proofs-of-concept. However, it is not robust or powerful enough to develop highly scalable, user-friendly, and secure desktop applications.

Encryption

We used the Cryptography and Fernet Python libraries to implement our encryption. As previously mentioned, these provided us with very simple and easy to implement encryption. As demonstrated on the documentation page for Cryptography, encryption with Fernet can be as simple as follows:

```
>>> from cryptography.fernet import Fernet
>>> key = Fernet.generate_key()
>>> f = Fernet(key)
>>> token = f.encrypt(b"my deep dark secret")
>>> token
b'...'
>>> f.decrypt(token)
b'my deep dark secret'
```

Figure 6. Simple implementation of Fernet encryption

This is great! But we decided to take it a step further for self-educational purposes. This requires entering the hazardous materials layer of Cryptography, which is prefaced by the warning:

“You should ONLY use [this layer] if you’re 100% absolutely sure that you know what you’re doing because this module is full of landmines, dragons, and dinosaurs with laser guns.”

We admit that we were not 100% absolutely sure of ourselves, but we viewed this as an educational exercise, and there was no fun in the simplest version. Note that any time we implement encryption and decryption, we could have alternatively chosen the method demonstrated above. Key generation occurs in three functions in Open Sesame: `add_service`, `generate_master_key`, and `generate_user_table_key`. Encryption occurs in `create_user`, `add_service`, `get_service`, and `update_service`. Encryption and key derivation also occur in authentication, but we consider this process separately from the rest.

Open Sesame works by first generating a master key, also called a Key Encryption Key (KEK) from the user’s master password. The user *must* remember this master password, it is not the responsibility of Open Sesame to store it. The key derived from this is then used in a key

derivation function to derive the user table key - the key that will be used to encrypt all information stored in the user table. The user table stores a user's KEK, user table key, and a validator used for authentication. The validator is the user_table_key encrypted on itself. Once the user has been created, i.e they have generated a master key and user table key, they can begin adding service/password pairs to Open Sesame. The service, username, password, and encryption key are all stored in the user's service_table, where the password and key are each stored as encrypted information. The password is encrypted with the key, and the key is encrypted with the KEK.

Key generation in generate_master_key uses a constant time key derivation function to derive a key from the user-created master_password. This uses the SHA256 hashing algorithm to generate the key. Generate_user_table_key uses key-based key derivation, which derives a cryptographically secure key from another key. The master password must first be encoded into URL-safe base64 bytes before it can be used to derive the key. This conversion from string to bytes became extremely important to take note of when it came to entering and retrieving information from our database. The functions described can be seen below.

```
def generate_master_key(master_password):
    ''' generates a master_key used for verifying user authenticat

    otherinfo = b"concatkdf-example"
    ckdf = ConcatKDFHash(
        algorithm=hashes.SHA256(),
        length=32,
        otherinfo=otherinfo,)

    safely = str.encode(master_password)
    master_key = base64.urlsafe_b64encode(ckdf.derive(safely))
    return master_key
```

```
def generate_user_table_key(KEK):
    ''' generates a key to encrypt values added t

    # key-based key derivation
    kdf = KBKDFHMAC(
        algorithm=hashes.SHA256(),
        mode=Mode.CounterMode,
        length=32,
        rlen=4,
        llen=4,
        location=CounterLocation.BeforeFixed,
        label=b"KBKDF HMAC Label",
        context=b"KBKDF HMAC Context",
        fixed=None)
    key = kdf.derive(KEK)
    return key
```

Figure 7. Key derivation for KEK and user_table_key

Encryption in `add_service` took the naive approach offered by Fernet. It generates a key using the Fernet `generate_key()` function, which returns a cryptographically secure key, which is then used to encrypt the password for the new service. The KEK encrypts the generated key before it is sent to the database, where all of this information stays at rest. Below are examples of the encryption in `add_service` and the decryption in `get_service`. Similar processes are followed in `update_service`.

```
add_service(service, username, password, KEK):
''' add a login (service, username, password) to be

key_KEK = Fernet(KEK)
key = Fernet.generate_key()
encrypted_key = key_KEK.encrypt(key)
str_en_key = bytes.decode(encrypted_key)
f = Fernet(key)
byte_pass = str.encode(password)
encoded_pass = base64.urlsafe_b64encode(byte_pass)
encrypted_pass = f.encrypt(encoded_pass)
str_en_pass = bytes.decode(encrypted_pass)

# decryption
byte_en_key = str.encode(encrypted_key[0])
byte_en_pass = str.encode(encrypted_pass[0])
KEK = Fernet(KEK)
decrypted_key = KEK.decrypt(byte_en_key)
key = Fernet(decrypted_key)
decrypted_pass = key.decrypt(byte_en_pass)
password = bytes.decode(base64.urlsafe_b64decode(decrypted_pass))
```

Figure 8. Encryption in `add_service` and decryption in `get_service`

This leaves us with discussing authentication. We developed our authentication process from a description of authentication in “‘Secure Password Managers’ and ‘Military-Grade Encryption’ on Smartphones: Oh, Really?” Our authentication checks the user-entered master password against a validator, and a user is authenticated if the function properly decrypts this validator. We will show the function below then describe its steps.

```

encrypted_user_table_key = str.encode(encrypted_user_table_key[0])
encrypted_validator = str.encode(encrypted_validator[0])
byte_KEK = generate_master_key(master_password)
KEK = Fernet(byte_KEK)

table_key = KEK.decrypt(encrypted_user_table_key)
b64_UTK = base64.urlsafe_b64encode(table_key)
key_table_key = Fernet(b64_UTK)
validator = key_table_key.decrypt(encrypted_validator)

if validator == table_key:
    eKEK = str.encode(eKEK[0])
    decrypted_KEK = key_table_key.decrypt(eKEK)
    if decrypted_KEK == byte_KEK:
        byte_esk = str.encode(esk[0])
        decrypted_service_table_key = key_table_key.decrypt(byte_esk)
        return [True, decrypted_KEK, table_key, decrypted_service_table_key]

```

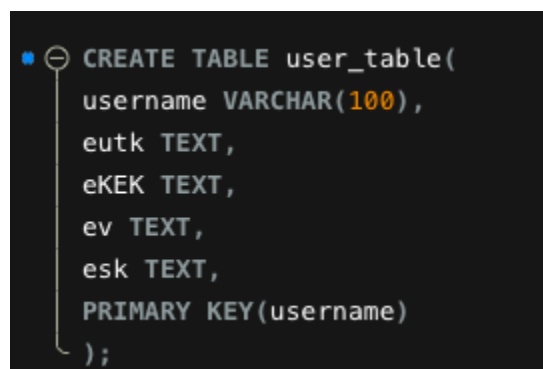
Figure 9. Authentication

First, the user must navigate to the login page, and enter their master password. This password is then passed to `generate_master_key`. If the user entered password is correct, `generate_master_key` should return the KEK we used to generate our `user_table_key`. We retrieve the `user_table_key` stored in the user table. We previously encrypted this key with the KEK, so if we have the correct KEK then `user_table_key` should decrypt correctly. Finally, we decrypt the validator with the `user_table_key`. The validator is the `user_table_key` encrypted on itself, so if we have the correct decrypted `user_table_key`, then the decrypted validator should be equal to the `user_table_key`. Thus, we have authentication! All that is left is a sanity check to make sure that the generated KEK equals the stored KEK, and then the decryption of necessary information.

Database

_____As mentioned in the Introduction, Open Sesame uses a MySQL database to store and manage passwords and sensitive information. In order to integrate MySQL into Python, we used the mysql-connector-python Python library to access and send information to and from our database.

Upon first use, Open Sesame users are prompted to create a name and password for their database, which is then the one used for their instance of Open Sesame on their local computer. From there, users receive their own user table, which stores their username, encrypted user-table-key, encrypted key-encryption-key, encrypted validator, and encrypted service-key. The schema can be seen in Figure 10.

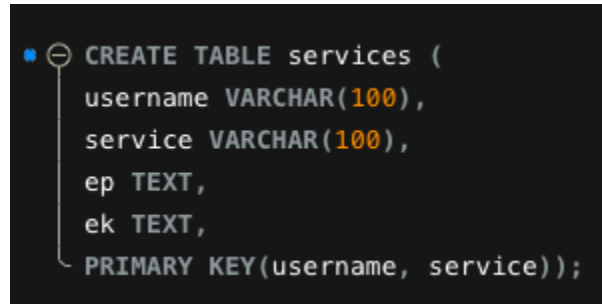


```
CREATE TABLE user_table(  
  username VARCHAR(100),  
  eutk TEXT,  
  eKEK TEXT,  
  ev TEXT,  
  esk TEXT,  
  PRIMARY KEY(username)  
);
```

Figure 10. “user_table” Table Schema

After the initial user_table is created users, once authenticated and after entering a master password, can begin to use the password manager like any other, adding, removing, and changing passwords. All of these services and passwords are stored in a services table, which holds their username, the name of the service, the encrypted password, and the encrypted encryption key. The exact schema can be seen in Figure 11. All of these functions require either retrieving or sending information to the user_table or the services table, which is done using the

Python connector and sending queries to the tables. In order to send information, it became crucial that we convert our byte-values into strings to ensure that they are being stored properly. To retrieve information, it was also crucial to ensure that we not only converted our string back to bytes but converted it properly- this would ensure encryption was working properly.

A screenshot of a SQL query defining the 'services' table schema. The query is displayed in a dark-themed code editor with syntax highlighting. It shows the creation of a table named 'services' with four columns: 'username' (VARCHAR(100)), 'service' (VARCHAR(100)), 'ep' (TEXT), and 'ek' (TEXT). A primary key is defined on the 'username' and 'service' columns. The query is preceded by a blue bullet point and a minus sign icon.

```
CREATE TABLE services (  
  username VARCHAR(100),  
  service VARCHAR(100),  
  ep TEXT,  
  ek TEXT,  
  PRIMARY KEY(username, service));
```

Figure 11. “services” Table Schema

When creating these predetermined queries that are needed for each function, our main concern was the threat of SQL injection attacks. In order to protect against these, we use prepared statements (Figure 12) in Python to ensure our inputs are sanitized before being sent to the database. These work by taking in the parameters as the type we designate them to be (in this case strings) so that malicious inputs will no longer read as SQL statements but as string literals, ensuring a level of safety against injection attacks.

However, a security issue presented by our database schema is that all of our primary keys are of a predetermined length. This means that if someone inputs a username or service that is over 100 characters long, Open Sesame will crash and the user will not be able to either access their manager or add/delete/change services. We would have liked to make these fields TEXT as opposed to VARCHAR(100) to combat this, but MySQL does not allow primary keys of undetermined lengths.

```
cursor.execute("INSERT INTO services (username, service, ep, ek) VALUES (%s, %s, %s, %s)",  
              (username, service, str_en_pass, str_en_key))
```

Figure 12. Prepared Statement Example

Future Directions

While we are proud to have built a password manager that successfully adds, removes, and changes passwords which are then encrypted and stored in a database, we have many areas to explore when it comes to future directions and Open Sesame.

Firstly, we would like to explore encryption in much more depth. Encryption is a major foundation upon which the security of our application relies, and as we previously discussed, we are not experts on it. Cryptography is an entire field of study, and we have merely scratched the surface. We implement encryption in Open Sesame that goes above the simplest scheme possible, but we would like to improve it further. A specific example of a place we can improve is our master key derivation. Sadly, we were not able to implement key stretching in our key derivation, and instead we had to settle for a key derivation function that runs in constant time. More broadly, if we had more time and expertise, we would have liked to increase the complexity of the key derivation we implemented, namely using more user secrets and more sources of randomness. Examples of user secrets would be account ID, user email, and secret keys generated upon user creation, which could subsequently be used to provide broader security. A similar point follows for more sources of randomness, which we were limited to by the functionality of the Cryptography library.

The next thing we would focus on is the database implementation and storage of the username/password pairs. Currently, using a MySQL database puts our password manager at risk of SQL injection attacks. As discussed, we sanitize our inputs before sending them to the database by using prepared statements, but there are many other countermeasures, such as character escaping and the concept of “least privilege” that we would have liked to implement to ensure the highest level of security. Furthermore, we also would like to possibly consider other options for our database system that are not a MySQL database. Moving past the locally deployed MySQL database that we are currently using could offer a more secure experience. As mentioned earlier, having a locally stored database comes with its own security issues, meaning that password managers may be more suited to have databases that are deployed on a server or cloud. Furthermore, our table schemas right now are quite revealing in terms of variable names, posing a big security risk if someone were to possibly gain access to the schema or the files.

Because of these risks, moving forward we would like to explore what it would mean to have our database be deployed as opposed to local, and maybe dive into what it would look like if we were to create and implement our own database management system for our password manager. Of course, a larger-scale deployment of our database would present a whole new host of security issues separate from those we face now, so deployment would take a lot of careful thought and consideration as well as careful design and implementation of the new database. The last place for future directions when it comes to our database is the fact that currently a user’s database is created upon first use of the app, with the database name, user, password, and host stored in a .env file. Since Open Sesame is an open-source password manager, having every user create their own .env file has potential security risks because some less experienced users may not know what a .env file is or how to keep it safe, as well as they may even accidentally push

their file to the GitHub repo. Also, if a hacker were to gain access to the .env file, they would have all the information they need to peer inside the central storage place of all of your passwords. While these passwords would still be encrypted, it poses a security threat, nonetheless. If we were to deploy our database, this issue would be resolved, and the database would be much more secure.

Moreover, in the future, we would like to focus on improving the overall user experience. First, we would like to improve our error handling and perform extensive testing on Open Sesame for more edge cases. For example, in its current state, if a user were to use Open Sesame and try to add a service that already has a password in the database, Open Sesame crashes and throws an error since that is a duplicate primary key error in SQL. It is apparent that this presents a huge availability issue- as the user will not be able to access their information because the application crashed. In the future, we hope to address this issue along with all of the other edge cases that we may have failed to think of because of the time restraint. Secondly, we would swap out the current images and assets for more suitable ones. For instance, when a user tries to use authenticated actions (i.e. add, get, update, delete) before successfully authenticating, the program displays a green checkmark and throws a success pop-up. Instead, since no operations were performed, this case should be updated to present a red x-mark and a failure pop-up. As previously discussed, many of our user experience issues could be resolved or better mitigated if we were to use a different GUI framework or library. While Gooey is a dead-simple implementation, it sacrifices flexibility, scalability, and a better user experience for its ease of use.

Lastly, we hope to package Open Sesame into an executable file that could be installed or distributed with minimal hassle. This would prevent users from having to follow our complex

setup instructions, such as installing and configuring SQL, installing dependencies with pip, and even having a Python interpreter installed. While Goocy does support packaging Python projects and their dependencies into executable files, we were unable to successfully utilize this feature.

Implementing these improvements would allow us to learn more about encryption, database security, and other concepts, furthering our knowledge and therefore making our product more secure. Given more time, we are confident that Open Sesame has the potential to become a relatively secure, usable password manager. Most likely never on a large scale like the Google Chrome extension or LastPass, but a lower scale such as a personal password manager.

Conclusion

Thus, it becomes apparent that one should only use a password manager that is designed, engineered, and maintained by a team of experts. It also helps if the password manager is well-received by security experts or if it is open-source, well-documented, and often checked by security analysts. The scale of this project and the experience of our team were limited, so we could not offer the robust suite of features that commercial password managers include. Moreover, we could not achieve the level of security which commercial password managers offer. In general though, one should not trust cryptography tools, libraries, or methods that they roll themselves. That being said, we entirely recommend others to pursue a project in password management or an adjacent topic in usable security. While creating a simple password manager with a GUI, encryption, and a database, we were faced with a slew of design decisions that could immensely affect user experience and security. This, in tandem with our research on secure design and engineering practices, proved to be an invaluable learning experience that beautifully demonstrated the complex and ever-evolving intersection between technology and security.

References

1Password Security Design. (2021). 89.

Belenko, A., & Sklyarov, D. (n.d.). “Secure Password Managers” and “Military-Grade Encryption” on Smartphones: Oh, Really? 12.

COMB: Over 3.2 Billion Email/Password Combinations Leaked. (2021, February 12). CyberNews.

<https://cybernews.com/news/largest-compilation-of-emails-and-passwords-leaked-free/>

Defending against crackers: Peanut Butter Keeps Dogs Friendly, Too | 1Password. (, 00:00 + UTC). 1Password Blog.

<https://blog.1password.com/defending-against-crackers-peanut-butter-keeps-dogs-friendly-too/>

Don’t trust a password management system you design yourself! | 1Password. (, 00:00 + UTC). 1Password Blog.

<https://blog.1password.com/dont-trust-a-password-management-system-you-design-yourself/>

encryption—Any hints for programming my own password manager? (n.d.). Information Security Stack Exchange. Retrieved May 12, 2021, from

<https://security.stackexchange.com/questions/198906/any-hints-for-programming-my-own-password-manager>

Gasti, P., & Rasmussen, K. B. (2012a). On the Security of Password Manager Database Formats. In S. Foresti, M. Yung, & F. Martinelli (Eds.), *Computer Security – ESORICS 2012* (Vol. 7459, pp. 770–787). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-33167-1_44

Gasti, P., & Rasmussen, K. B. (2012b). On the Security of Password Manager Database Formats. In S. Foresti, M. Yung, & F. Martinelli (Eds.), *Computer Security – ESORICS 2012* (Vol. 7459, pp. 770–787). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-33167-1_44

How Do Password Managers Work and Store Your Data? (2021, March 30). CyberNews.

<https://cybernews.com/best-password-managers/how-do-password-managers-work/>

Kerckhoffs's principle. (2021). In Wikipedia.

https://en.wikipedia.org/w/index.php?title=Kerckhoffs%27s_principle&oldid=1014779728

“Schneier’s Law”—Schneier on Security. (n.d.). Retrieved May 12, 2021, from

https://www.schneier.com/blog/archives/2011/04/schneiers_law.html