Swords: A Simple, Extensible, and Flexible Wallet-Based Password Manager File Format

Raden Rifqi Rahman - 13520166
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
radenrifqirahman@gmail.com

***Abstract*— Password manager is a software or program that stores our password securely by deriving our actual password using a master key. One kind of password manager is wallet-based password manager. Wallet-based password manager stores our secret inside a wallet file. Swords is a flexible and extensible file format for wallet-based password managers. Swords highly utilizes map or key-value pairs data structure. Swords stores a password or secret inside a record. Swords structures records into directory-like structure with the existence of collections. In consequence to swords' simplicity, flexibility, and extensibility; a password manager can extend swords format to provide more context or use minimal key-value pairs to save a lot of space.**

Keywords— swords, password manager, file format

I. INTRODUCTION

A password is the main protection of anyone's account on any services or products. Because of this specific use case, there is an issue when bad people, namely attackers or hackers, try to *hack* one's password to gain access to their account on a specific service or product. Although a lot of improvements have been made since the first scheme of password-based protection was used, e.g., the salting of password [1], there are still some problems with password-based security.

The most common problem of using password to secure one's account is not the security of the password itself or any kind of problem involving the software standpoint of the security. Instead, the problem lies within the human standpoint. As a human, we are limited to so much memory to remember everything, including passwords. Hence, two vulnerabilities arise when using a password for security, namely password reuse and weak password.

Password reuse is a situation where we use the same password for more than one account on various services. As stated, this is the case due to our nature of memory limitation. Oftentimes, we do not want to remember so much of a thing—passwords or string of characters—just to be authenticated or signed in to an account. As consequences, we tend to reuse the same password for multiple accounts on multiple services.

By the same reason, on the other hand, the vulnerability of weak password arises because passwords that are considered "strong" are often very hard to remember. For example, we might consider that a strong password would contain at least 12 characters which consists of at least one uppercase letter, one lowercase letter, one numeric character, and one symbol. Despite being strong, these requirements for a password result in a password that is not easy to remember. In other words, such a password is just a random string of meaningless characters. In consequences, we tend to use meaningful, relatable, and easy-to-remember words and numbers that are put together to be a password.

In existence of these two vulnerabilities, a software or program called *password manager* is developed to manage our password so that they do not need to remember any of their passwords. Such software, combined with a *password generator*, a program which generates random strong password, addresses most of the issues we discussed earlier. The password generator helps us generate a strong password, thus solving the weak password vulnerability, and at the same time also helps us to not reuse the same password repeatedly. On the other hand, the password manager helps us to store those various generated strong passwords, therefore disregards our human memory limitation.

While it sounds so simple, a password manager does not just plainly store our password in a file or a database to be retrieved in the future. A password manager secures our password by some transformations using a *master key*. Upon retrieval, the master key is then used to derive the actual password we store inside the password manager. Hence, we only need to remember one single password, the master key, to access all our password in the password manager.

Although the performed password derivation by the password manager is specific to itself and seems arbitrary, there is one similarity among any password manager. Regardless of how it operates to secure our password, password manager must store the secured password in one way or another. The most common way to do so is to store the secured password in a database, e.g., a key-value store, or a file. When stored in a file, a password manager must also store some metadata as general information which signifies the file as that specific password manager's file. Therefore, the file must have a specific format, whether an existing format such as JavaScript Object Notation (JSON) and Extensible Markup Language (XML), or a custom one. This piqued our interest to develop a file format which is specifically used for a password manager, along with an example password derivation scheme, with the goal of being a simple, extensible, and portable file format which we called *Swords*.

II. Terminology

A. Password Manager

Password manager is a software or program that stores our password securely by deriving our actual password using a master key. There are many schemes or variants of password manager classified by how they derive or store the password. Reference [2] explains that there are two main kinds of password managers based on how they operate, namely wallet-based password managers and hashing password managers. Wallet-based password managers work by storing our password in a "wallet" secured by a master key, while hashing password manager work by computing hash value derived by our password for an account on a site and the site domain [3]. Since we are developing a file format and password derivation scheme for a password manager, we refer to password manager as wallet-based password manager to be specific.

B. Master Key

A master key is the one key or password supplied to the password manager to derive the actual password. Master key is also often called master password, or the password-manager password. We refer to a master key as the key or password to access the stored password in our password manager file format.

C. Hash Function

Hash function is a function which receives a sequence of bytes as an input and returns the *hash value* or *digest* as a fixed length of bytes. We refer to hash function as any cryptographic hash function of any variations of properties, including block size, digest size, etc.

D. Encryption

Encryption is the process of transforming a meaningful piece of information into meaningless sequence of bytes using an encryption key. We refer to the act of encrypting and decrypting respectively as transforming the information into and retrieving the information back from a sequence of bytes.

There are two kinds of encryption in the field of cryptography, namely symmetric and asymmetric encryption. Symmetric encryption is an encryption where we use the same key to encrypt and decrypt an information, whereas asymmetric encryption is an encryption where we use a different key to encrypt and decrypt an information. To be specific, we refer to encryption and related terminologies as symmetric encryption since we do not use asymmetric encryption in our implementation.

E. Salt

Salt is a random sequence of bytes, preferably to be cryptographically secure, that is appended to a given sequence of bytes before it is hashed. Salt provides randomness property of the hash value of a given sequence of bytes. It is often used to secure password in form of hash value so that the digest of two identical value will be different.

F. File Format

File format is the content format inside a file—how the contents of the file, including metadata, main data, separators, and possibly magic number, are arranged in the form of bytes sequence. File format is also often referred to as file structure. A specific file format usually has its own file extensions, i.e., the suffix in the file name written after the last period character. We refer to file format as the format of the content inside the file as well as the file extension. Thus, we may use the word file format in the place of file extension.

G. Magic Number

We refer to magic number as a sequence of bytes that uniquely determine a file format. The magic number is usually placed at the start or at a specific position of a file having such file format. Some common file formats, such as PNG, JPEG, and PDF, have their own magic numbers which uniquely determine their respective file format.

H. Record

We refer to a record as a key-value store that is specific to context, which we shall see later referred to as label. In other words, a record is a *map*, that *maps* the key, which in this case is a string of characters to its respective value, which is a sequence of bytes. However, there are two kinds of value: plain value and secret value. Plain value is a value that is stored plainly, i.e., without transformations, in the password manager file. Secret value is a value that is securely stored in the password manager file using a similar password derivation scheme. To be specific, we refer to value as a plain value we defined earlier. In accordance, we will refer to secret value as a secret value.

I. Collection

A collection is a group of specific or similar things, either ordered or unordered. We refer to collection as a list of not necessarily ordered records. Additionally, a collection also stores some metadata in the form of a map similar to a record. A collection may also have children, i.e., collections inside such collections. Although it contradicts the definition of collection, we shall later see that our structure of collection provides a great feature in our password manager file format.

J. Map

Map is a data structure which maps a key of specific type to a value of specific type. In our design, we mostly refer to a map as a record-like map—a map that maps a string of characters to a defined value, either plain or secret.

III. Design and Methods

A. General File Structure and File Extension

To achieve a simple file format, we structure our file into three components. A *Swords* file starts with a magic number, followed by key-value entries header, and a collection which represents the *root* collection. Hence, our design of password manager file format always has at least one collection of records which is the root collection.

At the start of the file, we choose the sequence of the following sequence of bytes, written in base-16, as swords magic number.

```
73 77 6F 72 64 73 77 64
```

The choice of such bytes is not arbitrary. The sequence of bytes represents the string of ASCII characters "swordswd". Therefore, it suggests the idea of having our file format by looking at the ASCII representation of the first 8 characters in the file. The specific length of 8 characters is also not chosen arbitrarily. We chose 8 characters, having exactly the length of 8 bytes, so that it aligns nicely with various integer types. Thus, we can easily compare the first 8 bytes of a given file with our magic number as either 64-bit number, two 32-bit numbers, four 16-bit numbers, and so on.

The string "swordswd" also suggests the file extension of our file format. We chose the extensions ".swd" or ".swds" for our file format. The file extensions are chosen since it abbreviates our file format name, swords, and they are not well-known and much-used file formats.

After the magic number, we have the header of the file. The header is a map consisting of metadata of the file. Since it is a map, we may store as many entries as possible, depending on how they are used by the underlying password manager. We keep the entries flexible thus allowing a lot of possible extensions by the underlying password manager. However, we restrict the header to containing some preserved keys as we shall see in the later sections.

Following the header, we put a collection as the root collection of the file. Similar to its children, the root collection behaves the same as any other collection. The only limitation in our file structure is that no more than one collection can be a root collection thus it will disregard any remaining bytes after the first collection byte sequence ended. Therefore, swords file structure would look like the following.

```
<Magic Number>
<Header>
<Collection>
```

B. Entities and Starter Bytes

Before breaking down the header and collection structure, we introduce the concept of entities and starter bytes. To simplify all items contained in swords file, we generalize any item as *entity.* An entity is an item represented in an arbitrary length sequence of bytes which behaves uniquely based on their kind. This includes collections, records, and values. To uniquely determine the kind of entity, we put a unique starter byte at the beginning of its sequence of bytes. Simply put, an entity will have a structure as follows.

```
<Starter Byte>
<Byte Sequence>
```

C. Header Structure

Similar to a record, the header of swords file is a map of key to value. However, in contrast to a record, the header has its own restrictions on what keys it must at least contain. In our underlying password manager implementation, we require the header to at least contain the following information.

- Version, with key "v" and value of length 4 bytes.
- Master key hash function name, with key "mkhf" and value of arbitrary length.
- Key hash function name, with key "khf" and value of arbitrary length.
- Key cipher name, with key "kc" and value of arbitrary length.
- Master key hash, with key "mkh" and value of arbitrary length.
- Master key salt, with key "mks" and value of arbitrary length.
- Key salt, with key "ks" and value of arbitrary length.

Note that these information in the header is specific to password manager's password derivation scheme. Consequently, this restriction may not be applicable to other password managers. However, we encourage the underlying password manager to have at least an information of the version. We also suggest the key to be "v" as well to save some space as it is well understood that the letter "v" often stands for "version." While it should be clear that version must be a number, we did not standardize how it should be interpreted. Our underlying password manager may interpret the version as a single 32-bit number, whereas another password manager may interpret the version as two 16-bit numbers, or even having semantic versioning format with major, minor, and patch parts. In addition, we keep the meaning of version flexible. Our password manager recognizes version as the version of the file format, while another password manager may consider the version as the software or password derivation scheme version.

By having the general idea of how to interpret the header, we can now define the structure of the header. Since one of our goals is to make the file format simple, we structure every piece of information in the *byte sequence* portion of header entity as a key-value pair. To simplify things even further, we consider the *key* in key-value pair as also a value, i.e., it has the same structure as value which we will define later. This design is chosen due to the property of both key and value to have an arbitrary length by default. Therefore, the key in a map is essentially a special value with some restrictions. Such restrictions are as follows.

- Only plain value can be a key. Secret value cannot be a key.
- Key must be able to be decoded into a string in a chosen encoding, e.g., UTF-8.

All things considered, swords header structure looks like the following.

```
<"v"> <Value>
<"mkhf"> <Value>
<"khf"> <Value>
<"kc"> <Value>
```

```
<"mkh"> <Value>
<"mks"> <Value>
<"ks"> <Value>
<Key> <Value>
...
<Key> <Value>
```

D. Collection Structure

As stated, our form of collection consists of child collections, records, and metadata. It is highly analogous to a directory in a file system. As explained, we keep things flexible by using key-value map for nearly any information stored in the file. Accordingly, we use the same form of map to store metadata in a collection. Like the header, there are also some restrictions to the metadata map within a collection. However, we only require the collection to have a label with the key "label" and value of arbitrary length to name the collection to avoid confusing one with another.

Looking back at the header structure, we see that it may have an arbitrary number of key-value pairs with arbitrary length. Nonetheless, we also observe that the header does not store any kind of *length* information. That being the case, up until this point, we may assume that *Values*, as well as *Keys*, are *aware* of their own length thus allows the header to not store any length information. Furthermore, we determine that records are also aware of their own length. Using this flexibility, we can arrange the components of collection: children, records, and metadata, in an arbitrary manner. As an example, we may have a collection which is structured as the following.

```
<Record 1>
<"label"> <Value>
<Record 2>
<Child 1>
<Record 3>
<"some-data"> <Value>
<Child 2>
<Child 3>
<Record 4>
```

It is technically possible to have such structure of a collection. Nevertheless, we encounter a problem when we are to determine whether the entry "label" is part of "Record 1" or not. Another problem occurs when we encounter reach "Record 3" and "Record 4." Since collection also has an arbitrary length, it is not clear whether "Record 3" should be part of "Child 1" or the current collection, as well as whether "Record 4" should be part of "Child 3" or the current collection.

Recall that the collection is a form of entity which has a starter byte. In addition to the starter byte, we also add a terminator byte, specifically for collection, to avoid confusing record placement. As for the confusion of metadata that could belong to a record, we enforce metadata of a collection to be placed *before* any entities inside the collection—children and records. As a result, it is still possible to have mixed up and unordered placement of records and children within a collection. Nonetheless, the metadata will always be placed at the beginning of the collection up until the first entity is encountered. Having said that, our underlying password manager tries to keep the structure clean by placing child collections before any records. Moreover, the child collections and records are also placed in an ordered manner.

As for starter byte and terminator byte, we chose `03` and `04` respectively to represent the starter byte and the terminator byte of collection. Note that we cannot use the same byte as the starter and terminator byte because doing so will result in confusion whether we are encountering a child collection or terminating current collection when we came across the byte. In consequence, swords collections have the following structure.

```
03
<"label"> <Value>
<Key> <Value>
...
<Key> <Value>
<Child Collection>
...
<Child Collection >
<Record>
...
<Record>
04
```

E. Record Structure

Like header, swords record consists of key-value pairs. Since one record is intended to be used to store a single secret, e.g., a password of an account of a service; we require records to have a secret value of arbitrary length with the key "secret". As well as collection, we also require records to have a label with the key "label" to avoid confusion. Lastly, we define `02` as the starter byte of a record. In summary, a record would have the following structure.

```
02
<"label"> <Value>
<"secret"> <Secret Value>
<Key> <Value>
...
<Key> <Value>
```

Previously, we determined that records are aware of their own length. Still, it does not store any length information. Simply put, a record will contain all of the following key-value pairs up until a starter byte that is not the *value* starter byte encountered. In this sense, a record will contain as many key-value pairs as possible, thus is aware of its end—the byte before non-value starter byte.

F. Value Structure

Since value is aware of its own length and it cannot infer its end from a starter byte (a starter byte may be contained in a value), we need to store the length information of the value. In order to save space, we only use 2 bytes for length information thereby capping values at a maximum length of 65535 bytes. This choice is made because 1 byte of length information is too small since it would only allow values to have a maximum length of 255 bytes, while a hash function may output a digest with a length of 256 bytes, 512 bytes, or even more. Likewise,

3 or more bytes for the length information is too much. It is very impractical to have a single data or information to have a length of more than 65535 bytes. Therefore, 3 or more bytes for the length information would waste so much space if we store a lot of secrets or values inside swords file.

Recall that there are two kinds of value in swords file: plain value and secret value. To distinguish between two kinds, we could use a byte indicating whether the value is plain or secret. However, doing so would waste a lot of space as well. On the contrary, we have only reserved 3 bytes as starter (and terminator) bytes. In consequence, we can use different starter bytes for plain value and secret value. We chose `00` as plain value starter byte and `01` as secret value starter byte. As a result, a plain value is structured as follows.

```
00
<Length>
<Byte Sequence>
```

Likewise, a secret value is structured as follows.

```
01
<Length>
<Byte Sequence>
```

As an example, below is the sequence of bytes representing plain value of the string "hello".

```
00 00 05 68 65 6c 6c 6f
```

G. Authentication

Our implementation of password manager utilizing swords authenticate user using a master key. By storing master key hash function name, master key salt, and master key hash, we verify a given master key by converting the master key to byte sequence. Subsequently, the byte sequence is appended with master key salt, then hashed using the master key hash function resulting in a hash value. If the hash value is identical with stored *master key hash*, then the given master key is correct and user is authenticated.

The authentication using combinations of hash function, salt, and hash value is enough to verify the owner of the swords file. Nonetheless, we may observe that for the hash function, we stored the hash function *name* and not the function itself. That being so, in our implementation, we make a *hash function registry* which stores a map of hash function name to the function itself.

H. Secret Derivation

After the given master key is verified, we are finally able to derive all stored secrets from the master key. In our implementation, the swords header also stores key hash function name, key cipher name, and key salt. Instead of deriving the secrets, what we actually do is derive the key that is used to encrypt and decrypt the secret. As well as hash function, we make a *cipher registry* which stores map of cipher name to the encryption and decryption function.

To retrieve a secret, we retrieve the desired encrypted secret which is the value stored in "secret" key of a record. Afterwards, we derive the encryption key by appending master key byte sequence with key salt then hashing the result with the key hash function. With the encrypted secret and encryption key ready, we retrieve the decryption function from cipher registry then decrypt the encrypted secret using derived encryption key.

IV. Results and Discussion

A. File Size

Since we try to be simplistic and reserve space regarding swords file structure, we conduct a test on how big swords files are with various sizes of content stored inside. We pick four different cases of content size: nothing, no nested collection, one level nested collections, deeply nested collections. In addition to the content, we use the hash function SHA-256 for both master key verification and key derivation and AES256-GCM cipher for secret encryption and decryption for all four cases. Table I shows the contents and file size of each case.

TABLE I. Swords File Size

Case	Records	Collections	Nested Collections Depth	File Size
1	0	1	1	181 bytes
2	5	1	1	571 bytes
3	13	4	2	1228 bytes
4	21	7	4	1852 bytes

We observe that using SHA-256 and master key and key hash function and AES256-GCM as key cipher results in about 162 bytes of swords header size. Furthermore, by analyzing the results displayed on Table I, a record with a label about 6-8 characters long is about 80 bytes in size. Note that these 80 bytes include the extra information of "nonce" required to encrypt and decrypt data using AES256-GCM. Overall, it is evident that swords require only a tiny amount of space to store our password securely. As a comparison, with a storage space of 1 MB, swords is able to store about 12500 records. That is highly likely to be significantly more than we ever needed. Moreover, we could also store some binary data or information as a value in a record to provide more context. As an example, we may want to attach an icon or image representing the service that we store the password in the record for.

B. Password Derivation

Using the secret derivation scheme we explained earlier, we conduct a test to ensure that stored secrets are encrypted and decrypted correctly. For test purposes, we store a secret string "swordsswd" into a swords file secured with "swordsswd" as master key. Our implementation then generates random key salt with a length of 16 bytes and stores them in the header with key "ks" as we discussed earlier. By inspecting the swords file, we obtained the following byte sequence as key salt.

```
8B D9 43 C4 51 3F 36 54
9B 85 E1 5B F8 B9 5D E8
```

Upon creating a new record to store the secret, our implementation also handles the 12-bytes nonce generation for the secret. We obtained the following 12-bytes nonce.

```
EB CF 1B 9F 8B D5 23 71 ED 01 D7 4B
```

Along with the nonce, we obtained encrypted secret as follows.

```
3C 2E E8 CD 2C 29 DE 51 2D A4 9E 8A 8B 40 89 45 AC
51 9D E8 99 A2 F9 11 2E
```

To test the key derivation and decryption, we authenticate ourselves using the master key "swordsswd". After our password manager verified our master key, we head towards the test record inside the swords file. To avoid leaks, our implementation copied the decrypted stored secret into OS clipboard. By looking at the contents of our clipboard, our implementation correctly derived encryption key and decrypted stored secret back into "swordsswd".

V. Conclusion

Password manager is a software or program that stores our password securely by deriving our actual password using a master key. There are two main kinds of password managers based on how they work, wallet-based password managers and hashing password managers. Wallet-based password managers store secured password inside a wallet file to be retrieved when needed. Hashing password managers hashes the master key with external context or information to obtain the password.

Swords is a file format used for wallet-based password managers. Swords highly utilizes map or key-value pairs data structure to provide high flexibility and simplicity. Swords file consists of several entities: collections, records, and values. There are almost no restrictions on how a password manager interprets the key-value pairs inside a record in a swords file, thus providing high extensibility.

An example password manager which utilizes swords file format to store secrets is implemented using minimal keys stored in a record. It only needs the required keys, label and secret, and an extra key, nonce, in a record to be functional. It utilizes swords header to stores needed metadata for password derivation. It derives the stored password by appending salt bytes into master key, then hashes the resulting byte sequence into an encryption key. Using this encryption key and chosen cipher in the header, it decrypts encrypted password back to its original value.

Acknowledgment

First and foremost, I would like to praise and thank God, the Almighty, who has granted me countless blessing, knowledge, and opportunity to finish this paper. I would also like to express my gratitude to the lecturer of IF4020 Cryptography, Ir. Rinaldi Munir, who encouraged me to write this paper in the first place. Last but not least, I would also like to thank my family who supported me throughout my life.

References

[1] R. J. T. Morris and K. Thompson, "Password Security: A Case History," *Commun. of the ACM*, vol. 22, no. 11, pp. 594-597, Nov. 1979, doi: 10.1145/359168.359172.

[2] E. Stobert and R. L. Biddle, "A Password Manager that Doesn't Remember Passwords," in *Proc. 2014 New Secur. Paradigm Workshop*, 2014, pp. 39-52, doi: 10.1145/2683467.2683471.

[3] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell, "Stronger Password Authentication Using Browser Extensions," in *Conf. 14th USENIX Secur. Symp.*, 2005. [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/sec05/tech/full_papers/ross/ross.pdf