KeePassXC Application Security Review

Author: Zaur Molotnikov, zaur@molotnikov.de

The review can be updated in the future, locate the most current version, please, on https://molotnikov.de/keepassxc-review.

Document Versions and Changes

Version	Changes
v.1.2, 17.04.2023	Various text improvements based on readers feedback
v.1.1, 19.01.2023	Improved user recommendations
v.1.0, 18.01.2023	The first version intended to be published
v.0.3, 12.12.2022	Improved on definitions adding security, protect in memory feature analyzed
v.0.2, 07.12.2022	Included several attacks and paper reference, recommendations on changes
v.0.1, 05.12.2022	The first review of the core functionality

This document is an independent security review of the KeePassXC password manager version 2.7.4 functionality and central source code parts by me, Zaur Molotnikov, security consultant with applied security and applied cryptography basics knowledge. See my CV here: molotnikov.de/cv.

The review is provided for educational purposes, it is provided "as-is", without any warranties. I have prepared this review being an independent security specialist driven by my own interest in analyzing KeePassXC. My interest is namely caused by the lack of audited recently, free and open-source password managers, that I could myself recommend to a private person or to a company for use, knowing that I am recommending a secure solution in 2023.

The review is not associated, paid by or encouraged by my current or former employers, or any other party. This review is my best effort in scope of the limited hours effort. This is an ethical review, following this CISSP ISC2 code of ethics: https://www.isc2.org/Ethics. The review is formulated below to the best of my knowledge.

After writing this review, I have first shared it publicly with the KeePassXC development team, to give the development team a chance to decide if they want to revisit something or correct me, before I start publishing this review broadly. The follow-up conversation with the team allowed me to improve the review and even include a few new ideas. Many thanks to them.

Disclaimer. This review is not a recommendation or an endorsement. If you choose to use KeePassXC, you do so at your own risk. I do not guarantee any applicability, and do not warranty any protections that KeePassXC may or may not provide. You bear own responsibility and take risks on your own when selecting a particular technology, including KeePassXC, for use. All claims for direct or indirect damage that might be caused by your actions based on the materials presented here directly or indirectly will be rejected.

Total disclosure. I use KeePassXC myself and like the software. I do so, because I like Qt interfaces, C++, open-source software, and modern crypto. Am I emotionally biased to favor this program? -Yes! And this was exactly the reason, why I have decided to take KeePassXC apart, and analyze, and not another password manager that I do not like prior to analysis even.

I did my best to analyze this program *before* really liking it, recommending it, endorsing it. I think, the core problem of password managers today is one - they are not used enough. Thus, endorsing *some* password manager, especially the one I myself looked into, is a good thing to do, I believe.

Summary

KeePassXC provides sufficient cryptographic protection (confidentiality, integrity and authenticity) to the confidential information the user is storing in the database, given that the user selects a strong authentication method, e.g. a strong passphrase and a confidential random key file, and that the user will use KeePassXC with its latest secure file format.

The application is capable of reading and writing older and less secure KeePass file formats. Ideally, the application should warn on the use of insecure formats, suggest ways to migrate to the newest format. Should an attacker be able to replace the user's database with a database of an older format featuring the same authentication (this is a difficult for the attacker precondition), the user would loose authenticity and integrity of the information in the database. This is discovered by previous research, in the paper by Gasti and Rasmussen.

The password manager could also advise the user on improving protections, like selecting stronger KDF parameters once time passes by or by using protected attributes more often. The latter is relevant for a scenario, where a user might use a less secure password manager for the same database regularly. KeePassXC could store which latest version of the database was used by the user as well, and spot an undesired substitutions.

KeePassXC is written well and exercises defensive coding sufficiently. The memory deallocation could be improved to not to contain secrets after the database is locked though.

The key files must stay inaccessible to a potential attacker, as their authenticity is not checked, and digesting a key file might include complicated XML parsing, which has historically proven highly attackable.

I have reviewed the core features of KeePassXC focusing mainly on its database reading and writing features and the cryptography use. I could discover no major problems. This review, however, features a number of recommendations, that the development team could implement at their preference to keep raising security bar of the software.

As KeePassXC is a relatively complex program and the review effort was limited, I did not review all of the code base. Some helper features stay not reviewed, for example: TOTP, SSH agent, browser plug-in communication, auto-type, KeeShare password sharing mechanism, freedesktop integration, HIBP support, database statistics feature. Maybe these features could be a subject to a next review version.

To the best of my knowledge, disclaiming warranties and/or liability, I can recommend the use of core KeePassXC 2.7.4 functionality as of December 2022: reading and writing the database files with confidential user information.

Detailed Review

KeePassXC is a relatively complex application written in C++ programming language using Qt framework. The KeePassXC source code is approximately 127K lines large, excluding the libraries. The combination of the Qt framework and the application delivers a significant code base that is difficult to review as a whole. Yet, as the code is well structured, it was possible to review the core functionality independently of the rest of the code. I focus this review on parts of code relevant for encryption and storage of confidential information, the core functionality of a password manager.

In security profession practice, I have learned with time, that the main problem with password managers is that they are not used enough, not right or not regularly, not being perceived as easy to use and helpful software. To a regular user, a password manager, to be adopted well, should serve as a handy help to generate, store securely and retrieve conveniently their passwords. I personally find the user interface of KeePassXC appealing and recommendable, thus also my motivation to look under the hood, and know if it provides protection, that I could recommend as well.

I focus on a particular scenario, to also be able to consider the most central protection properties of the password manager and to not to deviate on other various and general attacks on computing as a whole (like side channel attacks on cipher implementations). The user will use the password manager on a trustworthy computer. The resulting encrypted password database, if presented to an attacker in an encrypted fashion, should be protected reasonably using cryptography selected by the password manager. In the course of the review, I explain and sometimes extend a little this context.

I leave out of scope scenarios, when the host may run not-trustworthy operating system, or where the host can be not trustworthy as hardware, be subject to environmental attacks, e.g. side channel attacks. These attacks, although realistic, challenge not only the password manager, but the software, where the passwords are going to be used, for example browsers. In our trust model, other software on the host, run by other users, and also software run by the same user might be not trustworthy, thus the password manager shall minimize the data exposed to the software running as the same user: the amount of data and the time window of exposure.

KeePassXC supports integration with browser extensions. The communication between the password manager application and the browser extensions is implemented using secure and modern libsodium-style encryption (I did not look into extensions yet past the encryption choice). I personally trust this cryptography choice and salut the use of encryption to communicate with browser extensions. It is worth noticing though, that being secure, libsodium encryption is not prescribed by standards like FIPS* as of now. Thus, when using KeePassXC in a high security environment where standardization of cryptography is mandated, I would recommend against the use of browser extensions. For private use, in my opinion, this is a very good choice of encryption. I do not review here in depth the implementation of browser extensions and communication with them though. It is a subject for the next review potentially.

Cryptography of KeePassXC relies on two solid pillars. First of all, it uses a rather standardized KDBX4 passwords database file format, which we will review below. Second, to implement the cryptographic primitives, KeePassXC relies on the existing crypto library, Botan, making a solid choice for it.

KeePassXC implements a cryptographic scheme which we will inspect below. It provides authenticated encryption combining AES256-CBC and HMAC-SHA256 to store users secrets. Argon2d is used as a key derivation function to produce a key from the passphrase. It is interesting to notice, that the selected master encryption for the database confidentiality and authentication is (as of now) quantum computer safe. I review the database format and the protections it provides below, also reviewing how KeePassXC is implementing it.

KeePassXC KDBX4 Database Format

KeePassXC is built to be compatible with older KeePass file formats. Here I will only review the new KeePass 4 KDBX file format. It is more secure than its predecessor, as it adds protected stream functionality and authentication to the database encryption. I recommend to every KeePass user to migrate databases in an older formats to KDBX4 latest format of KeePassXC.

The security of the password manager core functionality consists out of two ingredients: the inherit security of the format including the use of cryptography to achieve security, and the way the password manager implements the format. I review both here.

In all fairness, I would like to admit as well, that reviewing deeply and absolutely professionally an encryption scheme is a day job of a professional cryptographer. If someone of you know Paolo Gasti, Kasper Rasmussen, Bruce Schneier, Matthew D. Green, Steven Gibson, or another person of comparable caliber, asking them to double-check on my review could help. And yet, I believe, I understand and can explain well, why the core functionality of KeePassXC is secure below.

To my surprise, the KDBX4 file format is not described fully as a standard format, at least I could not find any descriptions more complete than these:

1. KeePass website discusses security features - web archive
2. KeePass website announcement - web archive
3. Gist from lgg - web archive
4. Gist from berlam - web archive

These descriptions also do not really feature any analysis on which protections are to be expected by various features of this complex, but, I believe, secure (details follow) and rather elegant format. I base my review on the source code of KeePassXC to avoid potential discrepancies in different descriptions of the format. These are some of the files under review:

1. KeePass2Reader.cpp
   - general logics to read the KeePass database files.
2. Kdbx4Reader.cpp and Kdbx4Writer.cpp
   - the reading and writing of the KDBX4 file format.
3. Database.cpp
   - database format, reading and writing features.
4. KdbxXmlWriter.cpp
   - functionality to write database internal XML format, compression for stored binaries, writing of protected entry fields.

Reading the C++ source code, I reverse engineer and explain the KDBX4 format, reviewing next its security features. In parallel, KeePassXC C++ implementation of the format is reviewed for application security typical flaws.

Long story very short, the database file consists of a public header and an encrypted body. The header is not encrypted and it does not have to be, containing public information only. The body is encrypted using AES 256 bit CBC encryption. The body and the header are authenticated with HMAC-SHA256. The user's authentication methods are securely transformed into keys for AES fore HMAC always combined with random bytes at each save to not to reuse the keys.

Additionally, the passwords inside the plain text for AES are encrypted with ChaCha20, called random stream here.

This format provides confidentiality and controls authenticity. Malicious or not integrity problem would always cause the database not to decrypt, there are no restore mechanisms and backups are needed.

Here is the layout of the database:

Bytes / Format	Field
8	File signature magic numbers
4	KeePass database version
variable	Database header fields:
+ 16	- Cipher choice
+ 4	- Compression: Gzip/None
+ variable	- KDF parameters
+ 32	- Master seed
+ m. cipher IV len	- Encryption IV
+ variable	- Custom public data
+	- End of header
32	Header hash (of the above)
32	Header HMAC
variable, body start	HMAC block stream, contains..

Bytes / Format	Field
+ variable	Main cipher stream, contains..
++ variable	Compressed stream, contains..
+++ variable	Inner header and XML Stream
+++ variable	Inner header:
+++ 4	- Random stream id
+++ 64	- Random stream key
+++ variable	- Binary attachments
+++	- End of inner header
+++ variable	Database XML, contains
++++ XML Tree part	Metadata of the database
++++ XML Tree part	Root group and its entries
++++ XML Tree part	Deleted objects

KDBX4 introduces some terminology, that will be useful when reviewing the format. I am presenting it below, relying on how mechanisms are called in the KeePassXC implementation. Please, note, inside the hash function formulas I use "+" operator meaning concatenation of bytes.

- Composite key - this is a SHA256-hashed concatenation of hashed incoming source keys that are used to protect the database: SHA256(SHA256(Passphrase) + SHA256(File Key) + . . .). See the implementation here. It is necessary to have the composite key to open/decrypt the database. The composite key is not yet protected by the KDF (by default Argon2d), see the transformed key and the final key definitions. The composite key is always the same for the database until authentication is changed by the user on purpose, e.g. by a passphrase change. The calculation of the composite key out of many inputs provides a mechanism for extending authentication ways with new ones. *Security*: The SHA256 relies upon Botan library.

- Compression flags - a 4 bytes integer of 0 for no compression, or 1 for GZip compression. KDBX4 may or may not use compression internally, prior to encryption, based on these flags.

- Compressed stream - if compression flags are set to 1, then this is a GZip archive of the XML stream containing the database entries, otherwise it is simply the XML stream without any compression. *Security*: Compression might identify password reuse. It will not happen by default with KeePassXC, as the passwords are encrypted with protected stream, see below.

- Database - depending on the context either the KeePass file, or the XML-encoded internal (decrypted) contents of it. KeePassXC stores a database as XML, encrypted with the master cipher.

- Database header - is located in the beginning of the database file and is unencrypted. It contains several fields in the format: 1 byte - field id; 4 bytes - field length n; n bytes - field content. The header contains ID of the main encryption cipher, usually AES, in UUID format; compression flags to indicate the use of gzip compression inside XML fields, master seed, Encryption IV, KDF parameters. The header includes as well public custom data, and the end of header field. The header is needed to know in which format and with which encryption the database was stored. *Security*: The header is designed to not to contain any confidential information, it is unencrypted. The header is authenticated with the Header HMAC and thus is protected from malicious or unintentional modifications. Shall the header integrity be lost, it will not be possible to decrypt the database with KeePassXC anymore. The user thus shall backup the database securely.

- Encryption IV - the IV for the master cipher, initialization vector for the AES256-CBC main cipher protecting the database. *Security*: The IV is never reused, at each save it is generated at random, using Botan's primitive for pseudo random or random, if available, numbers.

- Entry - an entry of a database, usually has at least these fields: a title, a user name, and a password, as well as creation time and possible custom fields.An entry can also contain binary attachments. *Security*: The password field of the entry is protected usually with the random stream. KeePassXC does not support ProtectInMemory attribute of entries. In other words, the protected stream is going to be protecting the fields at rest and during parsing or outputting XML, but not in the KeePassXC memory, when a database is unlocked. The logics to read the attribute is there, but it is not acted upon. The protection in memory comes from a Windows feature available through .NET to KeePass used in KeePass originally, that can use encryption to protect a fragment of memory to either same process or same logon. The protection provided by this feature is questionable, as only host administrator can read memory of any process in a good operating system, this means a compromised host, if an administrator is attacking the memory of KeePassXC maliciously. We do not consider such scenario in this research.

- Field - a part of an entry, e.g. a title field, or a password field. A named entity containing a value. *Security*: Can be protected by the protected stream, if the protected attribute is set to True. Passwords are protected by default. This ensures double encrypted at rest, using the protected stream and using the main cipher. The ProtectInMemory attribute of KeePass is designed to use CryptProtect feature to avoid plain text passwords in core dump files. KeePassXC does not make use of this field, but disables the core dumps.

- File key - a key that can be read out from a (secret) file and used to open the database. It is an optional mechanism, a user may choose to use or not to use a file key. The passphrase will still be needed to open the database. *Security*: File keys might feature XML structure and be parsed as XML, at the same time they are not authenticated. The user should keep the file keys confidential and secure, free from malicious modifications. File keys can be used by the user as a second factor authentication, e.g. by storing them on a USB flash and presenting the file as someone the user *has*, additionally to the passphrase that user *knows*. It is noteworthy, to say, that as the master seed is used in combination with the transformed key to produce the final master key, the file key is not strictly very necessary to further randomize the master key, in comparison to the use of the passphrase alone. Yet, the entropy is increased by using the file key too.

- File signature - KDBX4 files start with "signature" bytes, it is not a cryptographic signature, but 2 magic 4 byte ints 03D9A29A and 67FB4BB5 in the beginning of the file.

- Final key - or master key, the key for the master cipher, the secret (AES256 CBC) key protecting the confidentiality of the database. It is calculated like this: SHA256(Master seed + Transformed key) = SHA256(Master Seed + Argon2d(SHA256(SHA256(Passphrase)+SHA256(File key) + other authentication methods hashed)). *Security*: As the master seed is always generated at random at every save of the database, the final key will always be different. This prevents potential attacks on AES256-CBC when the same key is used to encrypt a slightly different message (incremental database versions). The final key is securely derived from the passphrase and other authentication methods the user provides to open the database. It is hard for the attacker to attack the final key, as the database does not contain information of all the authentication methods used, and attacking the passphrase is protected by Argon2d's KDF complexity.

- Group - a set of other groups and entries, a virtual folder inside the database. Database is organized like a tree by nesting groups and entries. Groups are created by the user and serve logical organization of the secrets purpose. *Security*: The groups can have properties that then can influence the corresponding properties of the entries: searchability, auto-type, expiration date. The user can use these features as they wish to regulate own security posture further.

- Header - same as Database header, an unencrypted portion of the database file located in the beginning of it. *Security*: The header does not contain any confidential information and is unencrypted. It is authenticated with HMAC, see Database header.

- Header hash - SHA256 hash of the database header data. It is written after the header to let a password manager verify header for not-malicious not intended corruption. Should the header be corrupted, the decryption process stops before even checking authenticity of the header. *Security*: The hash ensures integrity of the header, but not from malicious modifications, see the header HMAC. Should the integrity of the header be damaged, the database won't decrypt

- Header HMAC - HMAC-SHA256 of the header data, using HMAC key with the HMAC block stream transformation of it SHA512(UIntMax + HMAC key) = SHA512(UIntMax + SHA512(Master seed + Transformed key + \x01)). *Security*: Adds authentication to the header of the database file. In other words a malicious attacker will not be able to modify the header without being noticed. After the header hash is computed and is valid, header HMAC will be checked to verify authenticity of the database header. Should authenticity not hold, the decryption process stops. Should the header or the header HMAC be corrupted it will not be possible to decrypt the database with KeePassXC. The users should back up regularly. The key is different for every save being randomized by the master seed.

- HMAC key - one of the inputs to calculate keys used for the header HMAC and for HMACs of blocks of the main cipher stream alongside with the block index. The HMAC key is computed from the master seed and the transformed database key like this: SHA512(Master seed + Transformed key + \x01). *Security*: The master key changes between saves of the database, helping avoid HMAC key reuse, as the master seed is changing. The master cipher stream is split into HMAC-ed blocks. They are indexed and the indexes are used alongside the HMAC key to calculate keys for the block - to avoid reuse of HMAC keys on two different blocks. Only the authentic user can compute the HMAC key, as it is derived from authentication mechanisms used to store the database.

- HMAC block stream - A scheme for authenticated encryption with added data (AEAD). KeePassXC way to authenticate with HMAC the header of the file, and also to authenticate the encrypted cipher text of the database split in blocks. The cipher text produced by the main cipher is split into blocks of 1024 x 1024 bytes, then for each block an HMAC is calculated based on the transformed key, master seed and the block sequential number, see the HMAC block stream frame key. *Security*: It is an encrypt-then-mac scheme using the HMAC-SHA256 to achieve authenticated security. Should HMAC of any of the blocks of the cipher stream not verify, the decryption process of the database stops. This is another point motivating backups of the database.

- HMAC block stream frame - HMAC-protected block of data produced by the HMAC block stream. It has the following format: HMAC of the block - 32 bytes; length n of the block - 4 bytes; block contents - n bytes. The key that is used to calculate HMAC is calculated in increments, see the following point. The HMAC-ed frames add authenticated encryption to the main cipher stream.

- HMAC block stream frame keys - authentication keys for the database HMAC block stream. The keys for the HMAC function to calculate HMAC for the i-th block of data are calculated like this: SHA512(i + HMAC key) = SHA512(i + SHA512(Master seed + Transformed key + \x01)). The index i is represented as a 64 bit unsigned int. HMAC block stream frame key with i=UINT64_MAX is used as the HMAC key to compute the header HMAC. *Security*: The keys are based on the user authentication and are randomized by the Master seed. The database is authenticated often (1 KB of data) and with different keys for blocks.

- Inner header - a binary (not XML-encoded) encrypted by the master cipher start portion of the database data. It is followed then by the XML stream. Inner header contains fields in the format similar to the database header: 1 byte indicates a field id; 4 bytes indicate a field length n; then the field contents of length n follow. The inner header fields that KeePassXC recognizes are: inner random stream id, inner random stream key, binary inner header field, the end of inner header field. The binary inner header field - contains binary attachments for entries. These might be confidential files the user could store in a KeePassXC database. *Security*: The inner header is encrypted by the master cipher and is authenticated with HMAC. An attacker can not modify it without being noticed. The header contains the the inner random stream key material that will be hashed to form the key for the protected stream. This key material is encrypted with the master cipher as well.

- Inner random stream - same as Protected stream. A stream of pseudo-random bytes generated by the protected stream cipher, ChaCha20 in the default case of KeePassXC KDBX4 database format (but Arc4 and Salsa20 are also supported). This stream is used to encrypt protected entry fields. To encrypt/protect an entry with a protected attribute set, KeePassXC sorts the entries sequentially, and generates pseudo random bytes with the inner random stream xoring the protected field values with the generated pseudo random bytes. Example: the protected password field of the first entry is going to be xored with the first bytes of the protected stream, the next bytes of the protected stream are going to be used for the next protected field (usually a password again), and so on. Even after the master cipher was successfully used to decrypt the database, the decrypted XML will contain encrypted with the protected stream passwords, and not the passwords in the clear text. The protected fields after encryption are base64-encoded before storing to XML. *Security*: The protected fields enjoy a few indirect security benefits. First, as the values of protected fields are encrypted with random stream, even the same secrets get encrypted to different cipher texts, this prevents the compression stream from disclosing the degree of similarity of two entries, which might indicate e.g. a password reuse. Second, XML input and output facilities only work with encrypted data, shall there be memory problems in the XML facilities, the unencrypted values of protected fields will not be affected by them (e.g. leaked without secure free). KeePass used random stream encryption to cover for the absence of cryptographic memory protection functions in the operating system. This is weaker substitution for the native OS memory protection, as the attacker who could read encrypted values of protected fields, could also read out the protected stream key. If the attacker can only access some fragments of the application's memory, than inner random stream might help, as the attacker will never reach unencrypted protected values, and might never reach the encryption key. A powerful attacker able to read all application memory at any time will have chance to access the secrets always. A stronger protection, not compatible with other KDBX4 implementations though, could be to password-protect the protected fields. LastPass has such a feature.

- Inner random stream id - a 4 bytes number to identify the cipher used to produce the inner random stream / the protected stream. 1 for Arc4 variant, 2 for Salsa20, 3 for ChaCha20 that is used by default for writing databases in KDBX4 format. A field of the inner header.

- Inner random stream key - the key for the pseudo-random stream cipher, the protected stream. It is saved in an encrypted fashion as a field in the inner header. It is randomly re-generated at each database save. *Security*: The actual key is produced from the key material stored in the inner header. For the case of the ChaCha20 cipher, the key and the nonce are produced by hashing the material from the header with SHA512, and then taking the first 32 bytes as the key, and the following 12 bytes as the nonce, 44 bytes of entropy total. The key material in the header is always generated at random before the database is written. Thus the nonce and the key are always random for the inner random stream.

- IV - initialization vector for a cipher to randomize encryption algorithm. For example the master cipher IV is stored as encryption IV field in the database header, it randomizes AES256-CBC master cipher. *Security*: The master cipher IV is always generated at random and is not reused.

- KDF - key derivation function, the function that adds algorithmic complexity when generating decryption and authentication keys for the database. The default choice of the KeePassXC is Argon2d. *Security*: Argon2d is a secure PBKDF. A choice of Argon2id would be less prone to side channel attacks, that we explicitly consider below out of the attack model. The user might choose too weak parameters for Argon2d allowing for the database to open quicker, but also for the attacks on the passphrase to be more efficient. The default settings might also go up to the recommended in RFC 9106, especially growing the memory usage to a more representative for modern computers 2GB. KeePassXC reads older database formats and supports another KDF based on AES. The application could recommend a) to use Argon2d and b) to use sufficiently high settings for its parameters, see the RFC.

- Key file - a file, where the file key can be read from. It can have various formats. If no format is recognizable, KeePassXC would hash the contents of the file to obtain the key. *Security*: As key files are not authenticated, it is better not to store in non-trusted locations the key files, as the attacker might attack the complex parsing mechanism for the key files. See also the File Key.
- KeePass database version, KeePassXC database version, bytes 00000400 in my database. A magic number in the beginning of a database file.
- Master seed - public random bytes in the database header used to further randomize the encryption key for the main cipher, the random stream key and the HMAC block stream keys. These random bytes are always generated newly when a database is getting saved. *Security*: The master seed is added to the transformed key through concatenation and hashing, to then encrypt the database under different effective final master cipher key at each save. No mater the transformed key stays the same. It helps avoid final key reuse when re-encrypting and authenticating the database. This is an additional advanced protection feature is intended to avoid attacks based on the key reuse, for either AES256-CBC, or HMAC-SHA256.
- Master cipher - same as the main cipher, the symmetric cipher providing confidentiality to the encrypted database. By default it is AES 256 bit in CBC mode. *Security*: AES 256 bit CBC is secure choice for confidentiality. Authentication is provided by HMAC block stream encapsulating the master cipher stream. The IV and the key are always generated using randomness to avoid reuse.
- Master cipher key - see the final key.
- Metadata - of the database contains information about the database and some settings for the database as well. Information includes the name of the program producing the database, description, time stamps of changes, settings whether to use protected stream for titles, usernames, passwords, URLs and/or notes. *Security*: Metadata is a part of the XML stream, encrypted and authenticated with the master cipher and the HMAC block stream.
- Protected field - a field, that is encrypted using the protected stream cipher. This is encryption inside the value of an XML field in the database. Passwords are examples of protected field. They are xored with the protected stream before saving the value in XML. *Security*: See inner random stream.
- Protected stream - see inner random stream above.
- Protected stream cipher - Arc4 variant, Salsa20 or ChaCha20 cipher, working to protect XML field values of protected fields. See inner random stream. *Security*: see inner random stream. ChaCha20 is used by default for writing KDBX4 databases.
- Protected stream key - see inner random stream key.
- Root group - the top level group of the database where all other groups and entries belong, directly or through nested other groups.
- Transformed key - the composite key hashed (and protected) by the KDF, Argon2d by default: KDF(Composite key). *Security*: The key still has to be combined with the master seed to form the final master cipher key. This key is always the same for given database until changed by the user on purpose, that is why a random master seed is added to it to avoid key reuse for AES.

The key derivation from the user authentication methods is performed in a secure way, given that the user selects Argon2d and good parameters for it. The default parameters are okay, and are described as the second recommended choice in the RFC 9106.

- XML stream - same as Database XML, an encrypted inside the database stream with the contents of the user database, metadata, groups and entries.

With this we consider the encryption scheme and the file format described well enough, so that we can proceed to security analysis of it.

Security analysis of the KDBX4 format as implemented by KeePassXC

To analyze security of a particular encryption scheme, we need to define the context, and in particular, the attacker and the types of attack they could try to implement on the scheme.

Common context for the attacks

A user has confidential information, their secrets, including passwords and confidential files, and wishes to generate them to avoid reuse, and to store them securely across a number of trusted workstations, mobile and not. The user will use KeePassXC on one of their workstations.

To synchronize the database around the workstations, the use will use a [cloud] storage provider, which by assumption of this context, is not trusted. KeePassXC and other password managers under user's control will be storing the database repeatedly in a location, that is synchronized over network into to the storage provider's (attacker's) storage space.

The attacker is able to read all the saved bytes to their storage, change any of the bytes arbitrarily at their regard, time the writing and the reading operations. This is a higher attacking power, than the Advr and Advrw in this paper.

The attacker would like to determine as much information as possible about the confidential information of the user. This is an attack class targeting confidentiality of the information, Attack-C class, for confidentiality.

If possible, the attacker would like to sabotage the database as well, trying to change the confidential information available to the user after decryption. These are attacks on integrity and authenticity of the information Attacks-IAuth, for integrity and authenticity.

The attacker, as also non-malicious cloud providers, may lose parts of the encrypted database, risking availability of the confidential information, Attacks-A class, for availability.

The attacker would also like to attack the user by modifying the encrypted database file, as the input to KeePassXC application, in hopes to exploit KeePassXC and perform an online attack on the user. These are Attacks-E, for exploitation.

Finally, on the other workstations other password managers supporting KDBX4 database format might be used by the user. These password managers will not actively attack the user or the database, but, they might save the database using a weaker set of security features in a still compatible format. We will call this class of "attacks", Attacks-F, for format attacks. The attacker can notice the format downgrade and leverage it to their advantage, running one of the e.g. Attack-IAuth attacks that are available for the younger KeePass formats.

Assumptions:

1. The user is using a strong passphrase as their master password
2. The user, may or may not use other authentication mechanisms
3. The user stores their database versions incrementally, adding new secrets to the database
4. The user stores database backups in the attacker's storage

5. The user just uses KeePassXC, no other software is employed to verify authenticity or integrity of files
6. The attacker sees in real-time the writes of KeePassXC and can time the writes
7. The attacker can see and change any byte in the database

We play against the Advrw kind of an attacker, described in the paper by Gasti and Rasmussen.

Attacks on confidentiality

Protections of AES encryption

The confidential payload of the user resides encrypted in the KDBX4 file, inside the main cipher stream. The AES 256 bit CBC encryption provides confidentiality for the database. The main cipher protects the passwords, all other fields of the entries, attachments, deleted objects. Encrypted are as well the metadata of the database. In other words, no confidential content is stored in the database file without encryption. So a trivial Attack-C by reading the database file and having something in plain text won't work.

By design of our attack context the attacker, can observe multiple, in practice consecutive, versions of the database, the attacker could try to conclude something about the information stored in the database by comparing the versions.

The encryption of different database versions (incremental versions, backups) is performed always under a different key, final key randomized by the master seed, avoiding encryption key reuse. The IV is also random at every save, and is not reused for different database versions.

Database size attacks

The attacker might observe the size changes of the file. A reduction of size might mean permanent deletion of entries or attachments (the user is storing less secrets?). A growth in size might mean new entries or new attachments. As the attachments and the XML database are archived, the compression mechanism might reveal repetition of added data. This is because unique data added to an archive would usually increase the archive's size *more* than repetitive/already existing in the archive data of the same size.

The protected password fields of the entries will differ from one another before archiving, as the protected stream will encrypt even equal passwords to two different cipher texts (remember, it works sequentially on the protected fields). Repeated XML boiler plate will be archived efficiently, repeated not-protected entry fields will also be archived more efficiently, repeated binary attachments will also be compressed better. But the same passwords will not be compressed more effectively hinting an attacker that a password was reused.

Somewhat expected, but important to notice nevertheless: a type of Attack-C, where the attacker has to make conclusion on the size of the confidential data, the protection KeePassXC provides might not be same strong as similar protection by systems that first reserve a block of data, randomize it, and then it is more difficult to tell, how much of the data is actually confidential data, and which is just the result of the initial randomization.

Time attacks to measure the size of attachments and XML

I did not test this, but the binary attachments are saved without XML-encoding overhead, they are also recorded before the XML stream is even opened. Depending on how the buffering in the writing streams is organized, it might be possible for the attacker to observe a quick increase of the database file size until the XML is getting written, and then potentially slower XML addition. The buffering of writing streams, the networking to the not-trusted storage provider, both would make it more complex. But, depending on the skills, it might be possible for the attacker to reveal with time (through versions of the database saved) approximations of attachments sizes combined, and the size of the XML portion of the database.

As said above, I did not perform this experiment. This is also a very advanced and very targeted attack, only made possible by the context of network storage, it does not reveal the secrets the user stores in the database, so it is a very weak and also very theoretical attack. KeePassXC offers an alternative saving method writing the database in a temporary files folder first and then moving the temporary file into the place where the database should be saved, this should help avoid such attacks.

Attacks on the passphrase

The database file does *not* contain information on how exactly the composite key is generated: from a passphrase, or from a passphrase and a key file, or maybe with other authentication methods. Once at least one high entropy authentication method was used, the composite key starts to be hard to brute force. A random binary secret key file can be used for high entropy authentication, for example. Should the user choose to use a passphrase only, the attacks on the passphrase come into question.

Testing a passphrase might go through 2 ways: A) testing HMAC key with one of the HMACs, e.g. the header HMAC for simplicity and speed (the header is small); and B) testing AES key, e.g. by decrypting the first gzip stream block looking for magic numbers of a gzip stream.

In both cases, that attacker would need to generate a transformed key first. The attacker would need to hash the passphrase 2 times with SHA256, and then transform it with Argon2d.

For the case A), to get an HMAC key the attacker would need to combine the transformed key with the master seed, and hash it again with SHA512 receiving the HMAC key. Then HMAC key is combined with UInt64_Max and hashed again with SHA512 getting the header HMAC key. The test of the key has a complexity of computing an HMAC SHA256 of the header The complexity of this operation is ~ `2 x SHA256 + Argon2d(user parameters) + SHA512 + HMAC-SHA256` ~ `5 x SHA256 + Argon2d + SHA512` ~> `Argon2d`.

For the case B) The transformed key combined with the master seed is SHA256-hashed first. The test would include decryption of one block of the main cipher stream. The complexity of this operation would be ~ `3 x SHA256 + Argon2d + AES` ~> `Argon2d`.

In both cases testing a passphrase is slightly more complex than computing Argon2d hash of the passphrase. So the passphrase attacks boil down to Argon2d attacks.

As the encryption is randomized by Argon2d salt and by the master seed, it would be difficult to use a ready made rainbow table to attack the passphrase.

A smart dictionary attack is possible, should the passphrase be selected in a trivial fashion. Thus a requirement to the user holds: the user *must* select a long, complex, and sufficiently random passphrase for the passphrase-only protection to work well.

A brute force attack on the passphrase is even harder, it is harder than an attack on Argon2d.

Conclusion on confidentiality

The confidentiality of content is provided well by the main cipher, given that the user selects a strong authentication mechanism.

The attacker could conclude on the approximate length of the decrypted database. The attacker could also possibly conclude on the size of attachments and the XML portion of the database.

In practice, I can conclude, stating that KeePassXC implements KDBX4 format well and provides sufficient for daily use confidentiality protection.

Attacks on integrity

In this context the attacker is modifying available to them encrypted database breaking integrity and authenticity of the database. The password manager has to detect such attacks.

The integrity of the database is protected with the header hash, Header HMAC and the HMAC block stream's multiple HMACs for master cipher stream blocks. The attacker modifying the database would cause the decryption to fail safely. The authenticity check happens first on the Header HMAC, and then on the body HMACs, prior to decryption. This is good quality as the decryption algorithm will not be executed and attacked in this case.

Failing the authenticity check, the attacker destroys the database for the user, but can't manipulate confidential information protected by KeePassXC.

The HMAC authentication is happening without key reuse, as the HMAC keys are randomized with the master seed.

Shall the attacker substitute the most recent database version with one of the previous versions, KeePassXC and the user would not notice it, if the authentication mechanisms did not chain. The older version will decrypt normally, and the user will be presented with older version of the decrypted database. In this sense and in this context, KeePassXC is prone to replay attacks. It might be possible to store off-the-cloud the recent database versions, to prevent replay attacks.

In practice, it is difficult to imagine a good and realistic attack scenario. Maybe combined with some social engineering it could work, in situations where e.g. payment requisites stored in the wallet change, and the attacker revert the previous, no longer valid, version of them. Imagine a stolen bitcoin wallet address, that the user shall not longer use.

Replay attack remediation idea

This attack has limited impact, as the secrets change to the previous versions of secrets, the user themselves stored in the database, yet it is unpleasant. Let me suggest a potential and partial remediation:

1. Add to the database header custom public data, indicating file creation times (the attacker knows them anyway, it's not a problem to have it).
2. Before opening the database, show the contents of the custom public data, if the user selects so (should be optional, as here we parse unauthenticated data potentially)
3. Add and update a KeePassXC-authored meta data entry in the database's XML, store there information about the last time KeePassXC created the database, store there as well KDF parameters, the master seed and the master cipher IV (see later, why).
4. On open, after authentication, show to the user, when the database was really created last time by KeePassXC
5. Show to the user, once the custom public data and the KeePassXC authored entry are different, that another software was used to create the database, an older KeePassXC version not recording the creation times, or other password manager.

As a little bonus to user's security, noticing other password managers could also help the user recognize a potential substitution. Also, as other password managers might be less trusted or offering weaker format security, see KeePassXC as an example, the warning could serve security bar-raising.

Weaker password manager co-used attack

Above in the replay attack mitigation, we have already added a feature to detect potentially less secure password managers being used. Let's now mitigate the fact that they could be used by legitimate user, lowering the security bar, here are the mitigations:

1. Always offer upgrade to the latest database format.
2. Hard-code the minimal standard KDF parameters adequate at the time of the release, ask the user if they want to upgrade their parameters, if weak parameters are detected.
3. Detect, if password fields of entries stopped being protected, offer re-enabling protection.
4. Detect if another password manager reused the IV or the master seed, warn (first detect that another PM was used, then see that the IV or the MS are different).
5. Detect if another password manager lowered KDF parameters, warn.

Another way could be storing host-locally the last versions of databases saved. E.g. in Qt configuration files for the app. An upgrade for this would be authentication the settings.

Surely, the warnings could be optional, some users would not want to see them.

This mitigations is designed to detect weak, and not malicious PMs.

Format substitution attack

KeePassXC is designed for compatibility with older format version of KeePass. It openswithout warning or in fact without any visual indication the older format versions. The attacker could try to attack the user by replacing the authentic database by a rogue database of older format, the Attacks-IAuth and the exploiting Attacks-E can be attempted.

The older formats of the database contain a number of weaknesses, as described in this paper. In particular, the attacker can try to supply unauthenticated header of the database to the user, this substitution of the database will at first be not recognized by KeePassXC and decryption attempts will be made possible to the user, who will not suspect the replacement yet. This is the case where the decryption algorithm can be a subject for exploitation. The new database format prevents it using encrypt then HMAC scheme.

The attacker is in control of the unauthenticated header, but the user still inputs their old passphrase, which we assume the attacker does not know. The key generation algorithms, parameters of which the attacker now controls, unpleasantly, will process the users passphrase and produce a key, that the attacker can not predict, not knowing the passphrase. Moving on, KeePassXC will report invalid credentials used by the user. As it compares 32 random bytes in the start of the encrypted stream to the same 32 bytes stored in the header in the clear, stream start bytes, as they are called in KeePass. The attack will stop and not have any further dramatic consequences (but losing the database).

Running a decryption algorithm over not authenticated data, the inability to determine authenticity of the database, and the wrong error message (invalid credentials reported, whereas they are valid, the database is not), suggests a possible change of behavior to the KeePassXC program:

1. Warn the user, that they are opening an old database format that is poorly authenticated
2. Make the user take a choice, whether they want to open old and dangerous format
3. Do not write old formats by default after they are open, ask to upgrade the format right away, even before any features of the new format are used

The altered behavior will raise security of the user. It will be a choice of the user to let KeePassXC work with potentially poorly unauthenticated data, and not a default behavior.

Attacks on availability

On the premise that the attacker can modify the encrypted database, they can also delete it. Any modification of any byte of the database would lead to the decryption process failing for the user too, rendering the content unavailable to the user.

In such context, the user shall backup any database if they can not afford to lose its content. The backup shall happen in a separate independent and reliable location from the one potentially under the attacker's control. This is a trivia attack and defense case, by which KeePassXC could actively help, e.g. by offering a backup in another location to be automated, dealing thus with Attacks-A.

Backup wold also help the users in the case when the used storage medium is trusted, but is not reliable.

The key file should be authentic

This is not a weakness of KeePassXC file format, but rather a correct usage guideline, probably. A remark.

The key file should be kept secret, integrity of the key file should be maintained well.

The file containing the key can have multiple formats, including XML allowing for Attacks-E on the Qt XML parser.

The authenticity of the file is *not* protected by e.g. signature or HMAC based on the key, derived from the passphrase. It should be stored in an inaccessible for a potential attacker,trusted, secure location. File key shall be backed up, if the contents of the database must not be lost.

The use of the file key is optional, it increases security being a second factor added to a passphrase as the primary authentication mechanism.

Use of applied cryptography libraries

KeePassXC relies on the Botan library for cryptography.

Additionally, KeePassXC implements a key derivation function for older formats, based on AES. As we discourage the older formats and non-Argon2 KDF, I will not review the AES KDF here.

The Botan cryptography library when building KeePassXC on Linux is taken from operating system standard libraries. In my case, using Ubuntu Linux, the version `libbotan-2-dev amd64 2.12.1-2build1` was used.

The Botan library (in 2.4 version) was investigated and improved, and was recommended for use by the German Federal Office for Information Security (BSI): https://www.bsi.bund.de/DE/Themen/Unternehmen-und-Organisationen/Informationen-und-Empfehlungen/Kryptografie/Kryptobibliothek-Botan/kryptobibliothek-botan_node.html

Creating provable, certified, or anyhow other "good" software cryptography libraries is an art of its own, I do not dive deeper in the research here, leaving only my opinion following this sentence. The Botan library appears to be a solid choice, not being any worse or less standard than its competitors. The library is under development, the security advisory for it is available here: https://botan.randombit.net/security.html . There is no indication of relevant for the password manager use case advisories as of the time of writing this review.

Hash functions

KeePassXC uses HMAC functions based on SHA256 or SHA512 from the Botan library. It is a secure choice, used right in the database format.

Random numbers

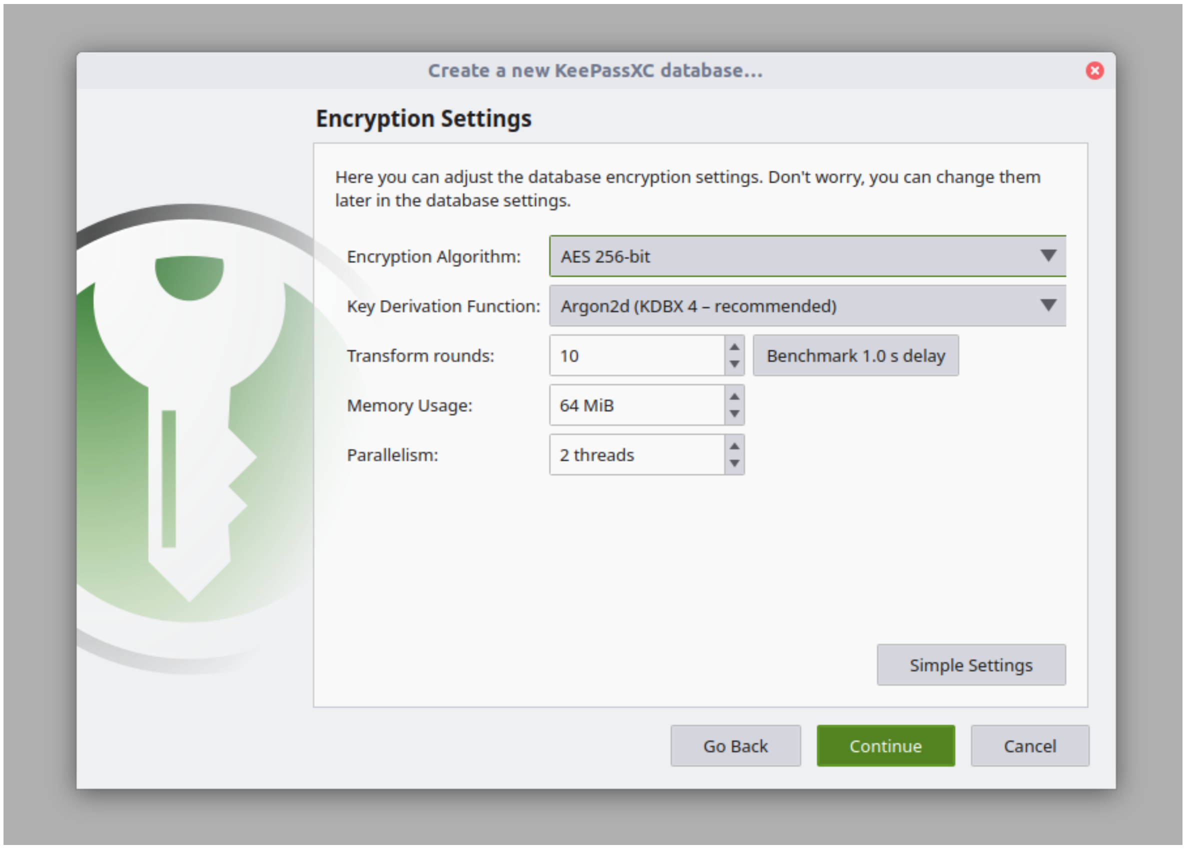
KeePassXC uses random number generator from the Botan library, which, if present, relies on the system's random number generator. It is a sound choice.

Symmetric ciphers

KeePassXC relies, again, on the Botan library to perform symmetric encryption and decryption.

By default, when a new database is created, KeePassXC defaults to a AES CBC 256 bit symmetric cipher for the master cipher choice.

Other options are TwoFish CBC 256 bit, and ChaCha20. All three options available are considered to be secure for the confidentiality use case as of December 2022, with the default choice, AES CBC 256 bit, being the most standard choice.



Create a new KeePassXC database...

Encryption Settings

Here you can adjust the database encryption settings. Don't worry, you can change them later in the database settings.

Encryption Algorithm: AES 256-bit

Key Derivation Function: Argon2d (KDBX 4 – recommended)

Transform rounds: 10

Memory Usage: 64 MiB

Parallelism: 2 threads

Figure 1: Default encryption settings

Various block cipher modes have advantages, disadvantages and extra-features. AES CBC 256 bit is secure if used correctly. It does not provide authenticated encryption. KeePassXC adds HMAC-based authentication to the AES CBC mode as discussed above.

Key Derivation Function

KeePassXC uses by default Argon2d KDF, the best choice of a password-based key derivation function as of now.

The implementation is included from an available C++ library on the operating system where KeePassXC is compiled. In case of my machine, it is the reference implementation by the authors of Argon2, Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich from University of Luxembourg.

Argon2id is considered to be marginally more secure, being less prone to side channel attacks. I recommend upgrading to it, although side channel attacks are not considered in this review a part of the attack model.

Applied Security Code Review

Code Quality

The code quality of security software shed light on application security too. Good structure and simplicity of code are necessary preconditions for security. KeePassXC has a well-organized, easy to review and to navigate clean code base. It is typical for Qt applications. Being a feature-rich GUI application though, this software can not be called simple. The code base features many test files. I did not find a quick way to measure the coverage, but given the amount of tests, 46 files and 20K lines of test code, I can conclude that the code is sufficiently well tested.

It is interesting to notice, that the code of KeePassXC is organized in classes cleaner than the code of KeePass in C#. The classes are smaller, the functions are shorter. See e.g. the code to read a database. The reason for this might be that KeePass grew organically into a more complex logical structure and maybe it could enjoy some refactorings. In any case, right now, it is easier to review KeePassXC code for security and to understand it than to do so for KeePass.

Defensive Secure Coding

Security software, like a password manager, should feature special coding techniques to protect the user's secrets and create additional layer of defense. KeePassXC uses a reasonable amount of defensive secure coding, and as any real-world software can get better at it.

There are two pillars of defensive coding really: checking the input and the output well, and managing memory well.

Reviewing the code where the databases are written and read, I could observe careful input checking, error handling, and clean, easy to review logics.

Memory Protection and Deallocation

This chapter is rather a reading material to induce own thoughts, I do not make any conclusion here, just some observations.

NSA published in November 2022 a paper talking about memory safety and encouraging a move to memory safe languages. The paper notices too, that even using memory safe languages, mistakes are still possible.

Software written in C++ can be prone to memory errors being close to C. At the same time C++ lets create better code with classes, reuse and the chance to call secure underlying operating system functions directly including precise and defensive memory manipulation.

It is easier to write maximally secure memory manipulating code when targeting one single operating system. KeePassXC is designed to work on Mac, Linux and Windows, for this it relies on Qt. QByteArray, for example, is used in many classes to perform cryptographic operations.

KeePassXC overloads the memory deallocation operators and includes secure memory scrubbing with Botan's secure_scrub_memory function.

KeePassXC makes effort to use secmem classes directly as well, example.

There is a point of uncertainty to me, that comes up if Qt libraries are used as dynamically loaded libraries, where the overloaded deallocation of memory might not play any role. It has been a long time I have coded in C++, but maybe an idea to a neat refactoring could be subclassing QString and

QByteArray and overloading their destructors and realloc functions, adding zeroing out or randomizing the occupied memory before just dereferencing it. The next problem to solve would be to make all the widgets use the new classes.. Compiling KeePassXC with the relevant parts of Qt statically starts being interesting.

Even after, we will not really be done. Internally used QXmlStreamWriter and -Reader would also have to use and deallocate memory in a secure way. Here we are with a parser and serializer in a security context. Securing them is not a task I would love to solve and guarantee the correctness of the results. Have a look at this rabbit hole.

The cost/benefit of protections by passing secure deallocation down to the every of the used libraries suggests leaving it as-is for now.

Essentially, what the use of normal data containers results into, is that the secret cryptographic materials could potentially leak, if a memory is deallocated but is not zeroed out. Using QByteArray for key materials, QString for the password and Qt Widgets, KeePassXC must rely on some built-in OS sanity, to not to leak cryptographic materials to other processes. Here we rely on the trustworthiness of the host machine and the host operating system.. And well, causing a holy war here, nobody serious, apart of Alexander Bluhm, cares to use OpenBSD as their OS daily. On other OSes *every* app will suffer from memory management problems and secure deallocation issues. Most interestingly, of-course, the browsers..

Actually, it is even *harder* to reach correct clean up of memory, if managed memory-safe languages are used. This remark is relevant for KeePassXC browser extensions, as well as to competitor products written in managed languages.

For the last part of this chapter, I would love to put some observations I made by performing core dumps of running KeePassXC 2.7.4 on my Linux machine and analyzing it. I was using gcore, strings and `grep` to see into memory of the application. Following are the results:

1. I could *not* see any of the passwords in the clear text in the dumps.
2. I *could* see parts of the database XML in the dumps, including user names and notes.
3. It was possible to see encrypted protected fields (see the format description too).
4. Unfortunately my notes in the entries were not protected, and I could read them in the clear.
5. It was also possible to see parts of the database XML *after* a database was closed.

The first 3 points are expectable. In the end of the day it is “just” a software that in the end of the day has to provide the user back with the information the user have stored in the database.

The last 2 points however do not feel really good.

I could see sensitive information in my “notes” fields, as I expected everything, and not just passwords, to be protected similarly well by KeePassXC. I never consciously made a choice not to protect the notes. This is an “unsafe” default with the KeePass family of password managers, in my opinion. I also have not found any UI to change this setting. It is possible to protect new attributes created for entries though.

This protection is not essential, but the code is there to have it, probably it could be easily made better. Of-course, encrypting more, might produce costs at other corners of the software. KeePass*DX* password manager decides not to protect even passwords by default, strangely, it is an option with KDBX4 format.

Some asymmetry in handling protected fields is also possible to observe in the entries search algorithm. KeePassXC can skip protected fields when searching entries. Yet it will only work for passwords and additional defined protected attributes. Notes, URLs, titles, user names - are all expected, in a hard coded way, to be not protected.

After the database is locked, it would be really nice and clean to not to have any remaining parts of it in the memory. KeePassXC makes an effort to disable core dump files. We also have to rely on operating system's protection of the application's memory, where memory of one application shall not be accessible to another one.

Secure Defaults

The most important defaults KeePassXC selects well are the encryption and KDF parameters selected well when creating a new database.

The main cipher is set to be AES256 bit CBC by default, a good secure choice for this application (AEAD is provided by HMACs additionally).

Argon2 is used which is the hash function of choice for passwords today. The parameters and the kind of Argon2 should be selected better as of January 2023, if we target the maximum security (not compatibility, and not the speed to unlock the database). By default KeePassXC selected Argon2d, 29 transform rounds, 64 MiB and 2 threads on my machine. The desired defaults should be Argon2*id*, 2048 MiB, 2 threads, and at least 1 round of transformation. Going for 4 rounds might be also adequate today, as modern computers solve it in under 10 seconds.

A note aside about the time benchmarking UI. In my humble opinion, the UI to calculate 1s delay is not great for Argon2, as it only increases the rounds of transformation. Instead, memory usage *and* rounds should be increased by it.

The password generator has a solid 32 characters default generated password length.

The algorithm to search entries has a feature to not to search protected attributes, like passwords. It is however disabled by default, so the passwords are going to be searched. This search is then implicitly propagated in the UI of KeePassXC. It makes not a lot of sense to me, to have a feature *on* by default to search (hopefully!) randomly generated passwords while searching for entries. I have noticed that my KeePassXC does not search password fields. But I do not see, where it is reflected in the code. Remember, other attributes are not protected by default in KeePassXC also increasing the search scope. I suggest a behavior change for this too, once a new field type is added to the entry - set the protected attribute to it by default.

Default choices (that can not be changed) to not to protect notes and some other entry fields are arguable as well. I do not know the reasons, why these protections are disabled by default.

I like the secure defaults of *not* switching the freedesktop secrets service, browser integration and sharing on.

Failing Safe

I not the biggest fan of this statement, relying on C++ left to right order of && evaluation. Now, what if it fails by writing many, but not all of the bytes in the file? Should the file not be closed and securely deleted instead of returning false?

Raising the secure removal bar of temporary attachment files, I wish they could enjoy some Gutmann voodoo, or at least a DoD method of 3 pass overwrites: one with zeroes with a check, one with ones and a check, one with a random character and a check. These are all old HDD tricks, SSDs would, of-course, like it less, showing more wear and causing TRIM to work potentially, moving the not-removed data in the shades of the SSD "surface". Surely, if a very resourceful attacker is considered, using full disk encryption, combined with VeraCrypt, combined with ATA Secure Erase and, naturally, physical destruction of the storage, mechanically, then chemically, is my non-paranoid recommendation.. (I wouldn't change the code, unless a use by the DoD is foreseen).

Validating Assumptions

Checking Input

Overall checking the input, in our attack scenario, is interesting in files reading the database. It is done reasonably well, I do not have any points to criticize here.

Networking

KeePassXC uses networking, if compiled so, for:

1. HIBP downloads
2. Update checks and
3. Icon downloads

As far as I understand the program and its code, the HIBP downloads work when a user request this feature from CLI, or from a special menu; KeePassXC warns that some of the information (first bytes of hashed passwords), will be sent to the HIBP service online. The icon download feature is also activated by user intentionally, when favicons are asked to be downloaded for entries. The update checker downloads a JSON file with release descriptions directly from GitHub API. These are safe and valid network communication patterns.

I would be cautious downloading favicons, if you are opening a database that you did not author.

By the way, documentation FAQ might be updated to state all the three scenarios.

Other input

KeePassXC reads configuration and words lists as well. Under the assumption of a trusted workstation, these operations are just fine.

KeePassXC uses Qt plugin system to support autotype functionality, reads and loads auto type plugins.

It is a wise idea to not to use KeePassXC on a workstation that the user does not trust, as this loading can compromise the software (and not only because of it).

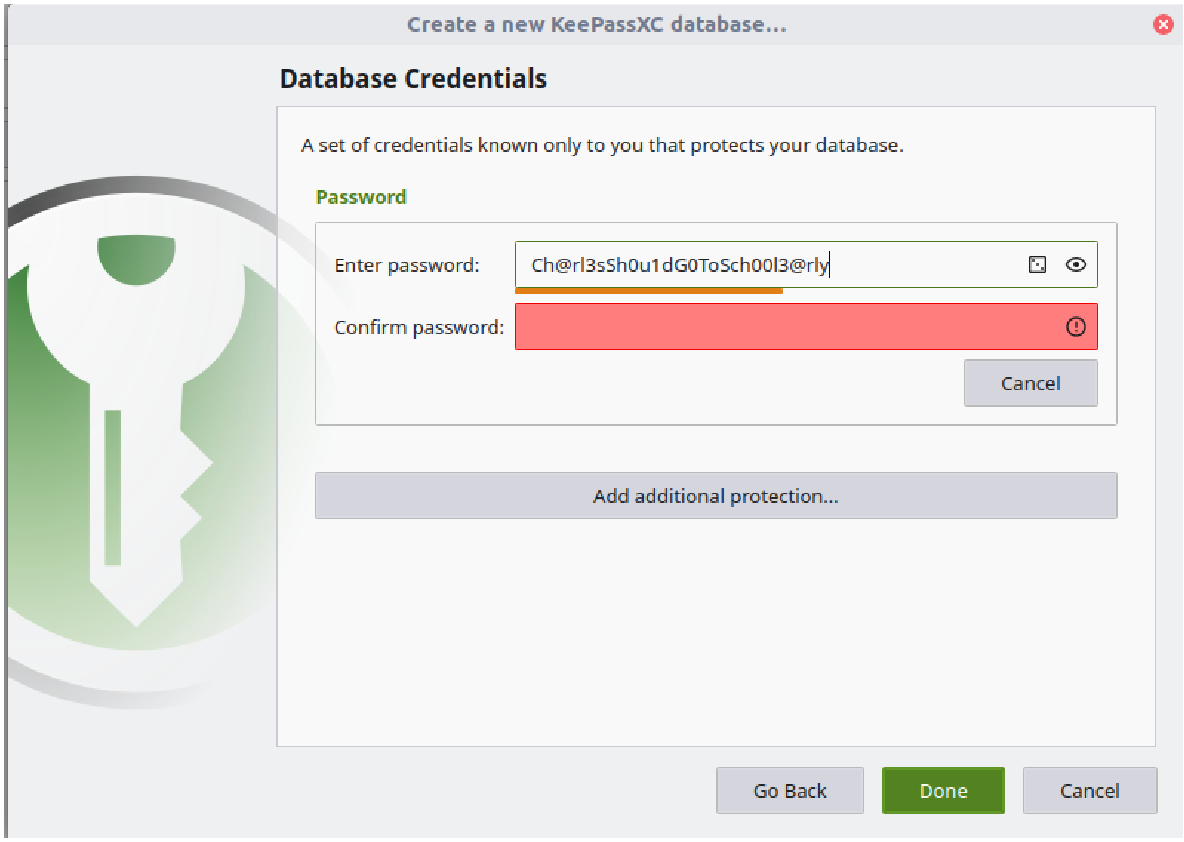
Additional Security Features

KeePassXC has a number of security features that can assist using the application in a more secure way.

KeePassXC by default does not show passwords, this is standard, but good feature, helping using the passwords without displaying them. It is possible to copy a password without showing it first.

To help protect passwords and database contents generally from an inadvertent screenshot or screen share, KeePassXC by default prevents recording of its window on Windows and Mac OS.

KeePassXC uses a C variant of Drobox'es zxcvbn library to check passwords strength and assists the user in assessing their passwords' quality, this is especially helpful for existing passwords that were not generated with a password manager. Zxcvbn is detecting various ways humans might introduce falsely "strong" passwords, and scores them low, indicating a mistake.



Create a new KeePassXC database...

Database Credentials

A set of credentials known only to you that protects your database.

Password

Enter password: Ch@rl3sSh0u1dG0ToSch00l3@rly

Confirm password:

Quick Unlock feature reduces the friction when using the password manager. On Mac OS and Windows the users can benefit from biometric sensors (Windows Hello, Touch ID) to quickly unlock the database. This makes it easier to use and lock more often the database, increasing security.

KeePassXC has a feature to backup the database at every save. This can help loosing important confidential information by accidental permanent delete. The recycle bin feature can prevent accidental permanent entry deletion. On an entry change a history record is created for the entry. It is possible to view the summary of a change, the old state of the entry, and it is possible to restore the old state. Should the history be unwanted, it's possible also to delete the historic versions.

Summary of recommendations to the implementation team

Here I suggest improvements to security that the implementation team could consider, in order of importance the way I perceive it. Pardon me the imperative, I use it for clarity, as user stories.

Urgent corrections of high-risk vulnerabilities:

None

Recommended improvements:

1. Detect, warn about and not open by default the old insecure database formats; insist on migration to newer formats.
2. Set Argon2*id* version of Argon2 KDF to be the default one.
3. Set default Argon2id rounds to t = 4 (1 at least), m = 2048 (at least), p = 2.
4. Color the simple complexity slider in red, yellow and green, similar to password strength meter, indicating where the user is in a danger zone.

5. In simple complexity UI, please, grow also the memory requirement, not just the t parameter for Argon2id.
6. Detect not-securely set KDF parameters, insist on improving them actively, warn the user (LastPass should be an example of what happens otherwise)
7. Detect the lack of protected fields, especially passwords, and insist on improving adding protected attributes.
8. Improve error-handling on the temporary files-writing code for attachments (secure delete, if partially written or not flushed).

Optional improvements raising security bar for the users:

1. Add an automated backup mechanism that would regularly backup the database in another location.
2. Keep a local, preferably authenticated, list of database versions, warn the user if the database was changed, describe the change, e.g. when and how the new database version was created, or that an older version is used (replay attack).
3. Consider the mitigations for weaker password managers used concurrently, described above.
4. Research the XML memory spillage further to know the exact reason for it, fix if possible.
5. Protect the notes of the entries by default.
6. Protect custom fields by default.
7. Implement a more standard secure delete algorithm, overwriting the files with 1., 1. and then random characters.
8. Migrate away from complex and unauthenticated key files, hashing a key file as a mechanism.
9. Bring the database reports and HIBP features further forward to the user's attention to help improve their security.
10. Warn the user to keep the key files confidential, and prefer not to store them in the same folder with the database, warn if it is the case.
11. Warn the user to backup the key files, as otherwise the database might be lost.
12. Protect the passwords in memory, so that the swap files would never contain plain text passwords, using OS native features.
13. If swap/core dump files are written by mistake - detect and (secure-) delete them on application start.
14. The advanced settings for KDF for existing database security setting is hard to find in the lower left corner. Make, please, this checkbox a well-visible button.

Recommendations to the KeePassXC users

When using KeePassXC, select a long and random passphrase always. It can not be overstressed, the complexity of the passphrase determines the level of protection.

Use the most recent database file format always. Check and migrate to the recent database format all of your databases.

Check your database security settings and change the Key Derivation Function like described in the following paragraph.

For higher security, when creating the database and selecting the parameters for the key derivation function, select Argon2id and then select at least 2048 MiB memory usage, at least 2 threads, and 4 transformation rounds (1 is sufficient, but 4 is better). Increase the parameters, if a little wait before the database file unlocks and at saves of the database is acceptable, it improves your security against passphrase attacks. Consider also that too high memory requirements might not work well if you use the database on weaker spec-ed old mobile devices. Note, this is less important than selecting a strong passphrase.

Use KeePassXC's feature to generate random passwords for you. Do not use the same or similar passwords on different websites. Configure KeePassXC to generate passwords that are at least 20 characters long. The high length should not be a concern for you, as you will be just using KeePassXC to remember and to fill in the new passwords.

Use KeePassXC's database reports statistics feature to view if you have weak, short or non-unique passwords still. Use "is:weak" in the search field to see which passwords might require a strong random replacement. Consider using HIBP functionality too to see if some of your used password appeared online before in the cracked database files. Note, please, these are useful features, but I have not reviewed their implementation yet, and can not yet vouch for their security. These both features are not really documented, you need to find them by selecting "Database" and then "Database reports" menu items. On the left you will see Statistics and HIBP.

Consider using a key file to improve the authentication security: the quality of the cipher keys will grow, an authentication factor will be added to the database too. Keep your key file in a secure preferably off-line way. Do not store your key file alongside the database. For example, store your key file on your host without synchronizing it to the cloud. Store your key file on a USB-flash, as a prove of having something.

Backup your database regularly, backup your database key files too. This is important if you store non-recoverable secrets like your crypto currency wallet seeds in the password manager. The password for most of the websites, on the contrary, usually can be recovered.

Consider splitting your secrets into more confidential and important, and rather less important daily-use credentials and store the secrets in two different databases. Do not use other, less trusted, password managers on your databases, or at least on the more confidential one. Note, one of your most confidential passwords shall be the password for your email box. Use second factor whenever you can in combination with your most important passwords.

KeePassXC documentation recommends storing your TOTP secrets in a separate database from the database where the passwords are stored (I personally do not follow this inconvenient advice, and use 2FA of my PM's passphrase and a key file).

And finally, read the KeePassXC documentation. It is short and good.

Existing Other Reviews

I could find a paper from Paolo Gasti and Kasper B. Rasmussen here discussing various attacks on password managers, and security of password manager database formats. It discusses KDBX4 files, but not the same database format version as KeePassXC is using. A different, older format, weaknesses of which have been addressed by the KeePass format innovation described here is the object of research in the paper. Gasti and Rasmussen start from the improvements in security in the KeePass 2 format. The improvements also hold in the KDBX4 format of KeePassXC. Continued, the weaknesses of the KeePass 2 format are described. I will go through the weaknesses noticed by the paper here below, one by one.

The first vulnerability described on the page 11, in the first two paragraphs, is the lack of authentication of the database header. Attacks are described next, that are possible based on not authenticated header. In the KDBX4 format used by KeePassXC the header is authenticated with header HMAC. The pskey parameter, described above as the Random stream key, in KeePassXC is a part of the body, it is encrypted by the main cipher stream and is authenticated by the HMAC block stream. The attacker does not have a chance to modify the pskey without breaking the decryption process of the database.

Side effects are discussed in the paragraphs 3 and 4 on the page 11. In the previous design, the pskey could also be modified without the user noticing it, by corruption or mistake of transmission. The led to a loss of passwords. With KeePassXC any modifications to a database would be detected, as the database = Header + Body; and both Header and Body are HMAC-authenticated. The attacker Advrw though has the chance to break the database decryption entirely. We address this destruction attack with an ask to the user to backup regularly off the non-trusted storage.

The paragraph 5 describes an attack that also relies on the, fixed now, lack of header authentication.

Reviewed Version

The version under review is the 2.7.4 release tag from Github.

I reviewed the source code located in the `keepassxc-2.7.4.zip` archive located here: https://github.com/keepassxreboot/keepassxc/archive/refs/tags/2.7.4.zip.

These are the SHA256 and SHA384 checksums of the archive.

```
3d45624f6000c665ecf40acac8733a053c7d68576568593debbae5cbbc5b84b2  keepassxc-2.7.4.zip
5aad7e8babf0ec7e40b90853d71b33427ce70454fcb2035dd19593d91cb0636..
  ..ac383d84e88a10f9f70ef9eec2cfa2680  keepassxc-2.7.4.zip
```

Review Effort

Here I am being transparent on how much time I have spent on this review, which should be an indicator, to some extent, of the depth of the review.

- 1 hour - start of the review for cryptographic primitives used and reproduction of the build process
- 1 hour - review of the standard symmetric cipher choice, review of other ciphers available
- 1 hour - review and write up of secure defensive coding in KeePassXC
- 3 hours - review of the KDBX4 database file format
- 2 hours - continued review of memory manipulations and KDBX4 format implementation
- 1 hour - experiments with the memory dump of the program
- 4 hours - continued review of the KDBX4 database file format and KeePassXC support for it
- 1 hour - confidentiality protection, revelation of sizes for attachments
- 1 hour - confidentiality protection, attacks on the passphrase, integrity attacks
- 1 hour - massaging the text for it to get readable
- 3 hours - reviewing other core files
- 3 hours - the draft 2 modifications, more attacks, review of database reading code further
- 2 hours - improving the text and the recommendations
- 1 hour - follow up on the protected attributes and protect in memory
- 1 hour - improving the recommendation, work on the text
- 1 hour - Argon2id vs Argon2d and parameter choice recommendation
- 2 hours - improving the text and the user recommendations
- 1 hour - improved the description of Attacks-C and complexity of key attacks
- 2 hours - improvements on text in the attacks sections
- 1 hour - communication with the team, improvements on recommendations
- 1 hour - recommendations are improved
- 1 hour - text and formatting improvements after the first readers feedback

33 hours total.

If you like my work and appreciate the effort, buy me a coffee here: https://paypal.me/qutorial

Hire me to perform a review like this for your product: https://molotnikov.de/contact